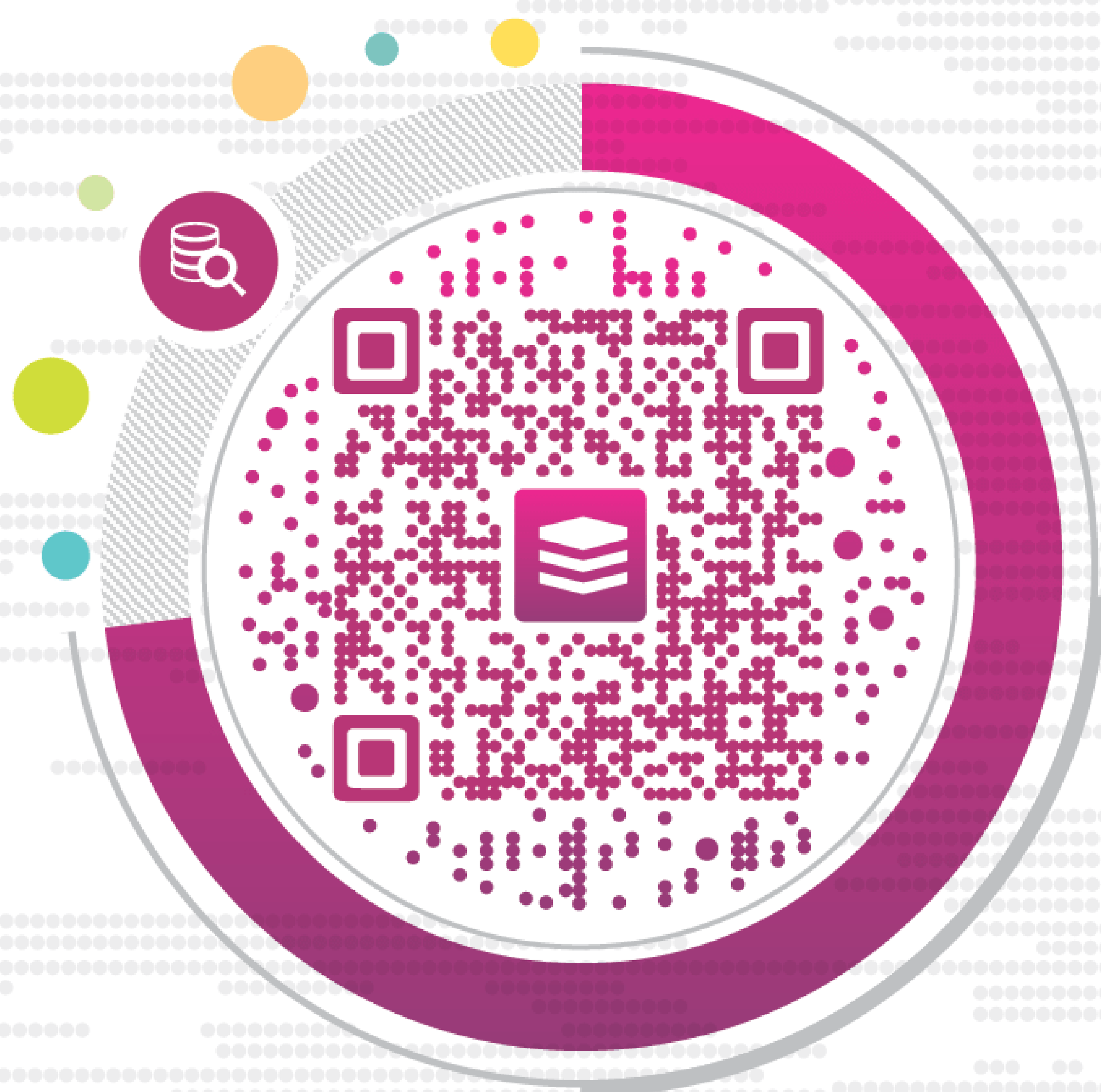
21世纪高等学校计算机类课程创新规划教材 · 微课版



新编数据结构案例教程(C/C++语言)

微课版

薛晓亚 主编　周丽平 马金霞 陈延波 副主编

21个综合案例　配套资源丰富

600分钟 63个微课视频

清華大学出版社

21 世纪高等学校计算机类课程创新规划教材 · 微课版

新编数据结构案例教程（C/C++语言）-微课版

薛晓亚　主编

周丽平　马金霞　陈延波　副主编

清华大学出版社

北　京

内 容 简 介

本书在主要介绍数据的逻辑结构、数据的存储结构、数据的运算等基本知识的基础上,从抽象数据类型的角度,讨论各种基本类型的数据结构及相关应用。

全书共分5篇:第1篇(第1章)为绪论篇,着重介绍数据结构的相关概念和算法的基础知识;第2篇(第2~5章)为线性结构篇,着重讨论线性结构的概念和基本运算的算法实现,介绍了一般的线性结构和特殊的线性结构在不同存储结构之下的基本操作和应用;第3篇(第6章)为树形结构篇,着重介绍基本的树形结构——二叉树在不同存储结构之下如何实现基本操作和应用;第4篇(第7章)为图形结构篇,介绍图形结构在不同存储结构之下的基本操作和应用;第5篇(第8~10章)为数据运算篇,首先介绍数据的查找和排序基本运算的算法实现,接着介绍常见的查找和排序方法,分析并对比它们的算法效率,最后介绍数据结构的基础知识在程序设计竞赛中的应用。全书提供了大量应用实例,每种算法都采用C/C++语言进行描述,帮助读者理解基础理论。

本书叙述清楚,便于教学和读者自学,适合作为高等院校计算机专业及信息相关专业的教材,也可作为计算机应用技术人员的参考书。

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。
版权所有,侵权必究。侵权举报电话:010-62782989 13701121933

图书在版编目(CIP)数据

新编数据结构案例教程:C/C++语言:微课版/薛晓亚主编.—北京:清华大学出版社,2019
(21世纪高等学校计算机类课程创新规划教材·微课版)
ISBN 978-7-302-51089-5

Ⅰ. ①新… Ⅱ. ①薛… Ⅲ. ①C语言—数据结构—高等学校—教材 Ⅳ. ①TP311.12 ②TP312.8

中国版本图书馆CIP数据核字(2018)第195636号

责任编辑: 刘 星 常建丽
封面设计: 刘 键
责任校对: 李建庄
责任印制: 李红英

出版发行: 清华大学出版社
网 址: http://www.tup.com.cn, http://www.wqbook.com
地 址: 北京清华大学学研大厦A座 **邮 编:** 100084
社 总 机: 010-62770175 **邮 购:** 010-62786544
投稿与读者服务: 010-62776969, c-service@tup.tsinghua.edu.cn
质量反馈: 010-62772015, zhiliang@tup.tsinghua.edu.cn
课件下载: http://www.tup.com.cn,010-62795954
印 装 者: 三河市君旺印务有限公司
经 销: 全国新华书店
开 本: 185mm×260mm **印 张:** 23.25 **字 数:** 564千字
版 次: 2019年7月第1版 **印 次:** 2019年7月第1次印刷
印 数: 1~1500
定 价: 59.00元

产品编号:077139-01

前　　言

一、为什么要写本书

随着信息处理技术和计算机技术的飞速发展，计算机在各个学科和领域得到了广泛的应用，而随着计算机处理的数据量迅速增大，数据类型随之增多，结构复杂的数据和数据关系的处理是我们必须面对的问题，由此设计一个结构好、效率高的软件，就必须分析并设计出好的数据结构，以便优质地处理数据的存储、数据传输和输出处理等操作。

"数据结构"作为计算机专业的一门核心基础课程，是计算机程序设计的重要理论和技术基础。通过本课程的学习，学生不仅可掌握各种组织方式下数据的存储、运算，而且还能使学生熟悉程序设计的基础方法，提高利用数据结构和算法解决实际问题的能力。

二、内容特色

本书有如下特色。

(1) 结构清晰、内容全面、文字描述简单明了、可读性强。

(2) 图文并茂，全书使用150余幅图描述数据结构概念、算法的基本思想、算法的执行过程。

(3) 强调数据结构中的3种逻辑结构和2种存储表示。

全书强调3种逻辑结构，即线性结构、树形结构、图形结构，对每一种结构都采用2种存储方式(即顺序存储和链式存储)表示，但必须注意每一种逻辑结构要结合其特点选择合适的存储方式表示。

(4) 由浅入深归纳总结每种数据结构的算法设计方法。

例如，利用创建单链表的算法，可以帮助实现链表的逆置、拆分、合并、排序等操作，利用二叉树的遍历算法，可以帮助实现查找结点、计算结点数量、计算二叉树高度、构造二叉树等。同样，图的很多算法都是基于遍历算法的。如果读者掌握了基本算法的设计方法，设计相关问题就容易多了。

三、结构安排

本书共分5篇10章，具体内容如下。

第1篇即第1章，为绪论篇。

第1章为绪论，主要介绍数据结构的基本概念、数据的存储表示和算法的时间复杂度等内容。

第2篇即第2～5章，为线性结构篇。

第2章为线性表(线性表是最基本的数据结构)，主要介绍了线性表的概念、线性表的抽象数据类型、线性表的两种存储结构(顺序表和链表)和常见的基本算法设计，并通过示例深入理解线性表的应用。

第 3 章为栈与队列,是操作受限制的线性表,主要介绍栈的概念、栈的抽象数据类型、栈的两种存储结构(顺序栈和链栈)和基本运算算法设计、栈的应用算法设计;队列的概念、队列的抽象数据类型、队列的两种存储结构(顺序队列和链队列)和各种基本运算算法设计、队列的应用算法设计。

第 4 章为串,是特殊的线性表,主要介绍了串的概念、串的存储结构、串的几种基本运算算法设计和串的模式匹配算法设计。

第 5 章为数组和广义表,主要介绍了数组的概念、数组按行和按列两种存储方式实现、几种特殊矩阵的压缩存储方式、稀疏矩阵压缩存储及转置算法设计;广义表的概念、广义表的存储结构及相关算法设计。

第 3 篇即第 6 章,为树形结构篇。

第 6 章为树和二叉树,主要介绍树的概念及逻辑表示、树的性质、树的存储结构;介绍二叉树的概念、二叉树的性质、二叉树的基本运算算法设计、二叉树的遍历运算算法设计(非递归方式和递归方式)、线索二叉树的概念和构造、哈夫曼树的概念和构造、哈夫曼编码构造等。

第 4 篇即第 7 章,为图形结构篇。

第 7 章为图,主要介绍图的基本概念和逻辑表示、图的存储结构、图的基本运算算法设计、图的遍历算法(DFS 和 BFS)及相关应用,尤其是工程上常用的最小生成树、最短路径算法、AOV 网及 AOE 网的基本算法等。

第 5 篇即第 8～10 章,为数据运算篇。

第 8 章为查找,主要介绍查找的概念、查找效率的度量标准。本章分为静态表的查找、动态查找表和哈希表查找,分析并对比各种查找方法的查找效率。

第 9 章为排序,主要介绍排序的概念、排序效率的度量标准,插入排序、交换排序、选择排序、归并排序、基数排序的算法设计,并对各排序算法的时间复杂度和空间复杂度进行分析和比较。

第 10 章为 ACM 经典案例,主要介绍以数据结构理论知识为基础的深化应用,探讨如何综合应用数据结构基础知识参与程序设计竞赛,增强学生的竞技精神。

本书的第 1、2 章由周丽平编写,第 3、5、6、7、8、10 由薛晓亚编写,第 4 章由马金霞编写,第 9 章由陈延波编写。全书由薛晓亚统稿。



源代码下载

四、读者对象

- 对数据结构课程感兴趣的读者。
- 计算机相关专业的本科生、专科生。
- 职业技术类院校计算机相关专业本科生、专科生。

五、致谢

感谢张秀国、高伟林、赵晓庆、田路阳、李冉冉等在本书的资料整理及校对过程中所付出的辛勤劳动。

由于编者的水平和经验有限,加之时间比较仓促,疏漏之处在所难免,敬请读者批评指正。

编　者

2019 年 2 月

目　录

第 1 篇　绪　论　篇

第 2 篇　线性结构篇

第3篇　树形结构篇

第 4 篇　图形结构篇

第 5 篇　数据运算篇

第1篇　绪　论　篇

第1章　绪　论

数据结构针对非数值计算的程序设计问题，研究计算机的操作对象以及它们之间的关系和操作，是介于数学、计算机硬件和计算机软件三者之间的一门核心课程。学习此课程的目的是了解计算机处理对象的特性，将实际问题中涉及的处理对象在计算机中表示出来并对它们进行处理。例如公交车线路问题，如何换乘车用最短的时间到达目的地，或者如何用最短的距离到达目的地等。本章是数据结构的基础，主要介绍数据结构的基本概念、数据的逻辑结构和存储结构、抽象数据类型和算法性能分析等基础知识。

1.1　什么是数据结构

1.1.1　数据结构的产生与发展

"数据结构"是计算机及相关专业的专业基础课之一，主要学习用计算机实现数据组织和数据处理的方法。它也为计算机专业的后续课程（如操作系统、编译原理、数据库原理和软件工程等）的学习打下坚实的基础。

随着计算机应用领域的不断扩大，非数值计算问题占据了当今计算机应用的绝大部分，简单的数据类型已经远不能满足需要，各数据元素之间的复杂关系已经不是普通数学方程式所能表达的了，且无论是设计系统软件，还是应用软件，都会用到各种复杂的数据结构。因此，掌握好数据结构课程的知识，对于提高解决实际问题的能力将会有很大的帮助。实际上，一个"好"的程序无非是选择了一个合理的数据结构和一个好的算法，而好算法的选择很大程度上取决于描述实际问题所采用的数据结构。所以，要想编写出"好"的程序，仅学习计算机语言是不够的，必须扎实地掌握数据结构的基本知识和基本技能。因此，在程序设计中，往往遵循"程序＝数据结构＋算法"的法则。

在了解数据结构的重要性之后，开始讨论数据结构的相关定义。本节先从一个简单的学生表示例入手，然后给出数据结构严格的定义，接着分析数据结构的常见类型，最后给出数据结构和数据类型之间的区别和联系。

1.1.2　数据结构的基本概念


视频讲解

1. 数据

数据是描述客观事物的数和字符的集合。例如，日常生活中使用的各种文字、数字和特定符号都是数据。从计算机的角度看，数据是指所有能被输入到计算机中，且能被计算机处理的一切符号的集合，它是计算机能操作的对象的总称，也是计算机处理信息的某种特定的

符号表示形式。例如,201702 班学生数据就是该班全体学生记录的集合。

注意:数据的范围是随着计算机的发展而不断扩大的。

人们通常以**数据元素**作为数据的基本单位研究数据。例如,201702 班中的每个学生记录都是一个数据元素。在有些情况下,数据元素也称为元素、结点、顶点或记录。一个数据元素也可以由若干数据项组成。

2. 数据项

数据项是具有独立含义的最小数据单位,也称为字段或域。例如,201702 班中每个数据元素(即学生记录)由学号、姓名、性别、籍贯、电话等数据项组成。数据对象是性质相同的数据元素的集合,是数据的子集。

3. 数据结构

数据结构是指所有数据元素以及数据元素之间的关系,可以看作是相互之间具有某种或某几种特定关系的数据元素的集合,即可把数据结构看成是带结构的数据元素的集合。数据结构包括如下三个方面。

- 描述数据元素之间的逻辑关系,即数据的**逻辑结构**。一般情况下,数据结构和逻辑结构是等价的。数据的逻辑结构是从逻辑关系(主要是指数据元素的相邻关系)上描述数据的,与计算机硬件无关。因此,数据的逻辑结构可以看作是从具体问题抽象出来的数学模型。
- 数据元素及其关系在计算机内存中的存储方式,即数据的**存储结构**,也称为数据的物理结构。数据的存储结构是逻辑结构用计算机语言的实现或在计算机中的表示,也称为映像,也就是数据的逻辑结构在计算机中的存储方式是物理结构,它与计算机有关。
- 施加在数据上的操作,即数据的运算。

表 1.1 中,整个学生表为学生数据对象,该数据表中包含 6 个学生记录信息,每一行学生记录(例如,张三的记录信息)代表一个数据元素。张三的记录信息中包含学号、姓名、性别、籍贯、电话 5 个数据项。

表 1.1　学生数据表

学　号	姓　名	性　别	籍　贯	电　话
01	张三	男	北京	666301
02	李四	男	南京	666302
03	王五	男	北京	666301
04	赵六	女	郑州	666303
05	钱七	女	青岛	666303
06	孙八	女	郑州	666306

}一个数据项

一个数据元素

为了更确切地描述数据结构,通常采用二元组表示。

```
Data_Struct=(D,R)
```

其中,Data_Struct 是一种数据结构,由数据元素的集合 D 和 D 上二元关系的集合 R 组成,即

$$D=\{d_i \mid 1\leqslant i\leqslant n,\ n\geqslant 0\}$$
$$R=\{r_j \mid 1\leqslant j\leqslant m,\ m\geqslant 0\}$$

其中，d_i表示集合 D 中的第 i 个数据元素（或结点），n 为 D 的基数，即数据元素的个数。特别地，当 n=0 时，表示 D 是一个空集，因而 Data_Struct 也就无结构可言，有时把这种情况认为是具有任意结构。r_j表示集合 R 中的第 j 个关系，m 为 R 中关系的个数。特别地，当 m=0 时，表示 R 是一个空集，表明集合 D 中的数据元素间不存在任何关系，彼此是独立的。

1.1.3 逻辑结构的种类



视频讲解

在不会产生混淆的前提下，常常将数据的逻辑结构简称为数据结构。根据元素间的关系，将数据的逻辑结构分为以下 4 类。

1. 集合

集合是指数据元素之间除了“同属于一个集合”的关系外，别无其他关系。例如，整数集合、字母表集合。

如图 1.1 所示，元素 A、B、C、D、E 除了同属于一个集合外，相互孤立，无其他关系。

2. 线性结构

线性结构是指该结构中的结点之间存在一对一的关系。其特点是：开始结点和终端结点都是唯一的，除了开始结点和终端结点以外，其余结点都有且仅有一个前驱，有且仅有一个后继。线性表就是一种典型的线性结构。

如图 1.2 所示，线性结构中每个数据结点有且仅有一个前驱结点（除第一个结点外），A 结点为第一个结点（首元结点），B 的前驱结点为 A，C 的前驱结点为 B，其他以此类推。这种数据结构的特点就是结点之间存在一对一的关系，即线性关系，因此是一种线性结构。例如，表 1.1 中每一行记录之间的关系，或者生活中在食堂排队打饭，在银行排队办理业务等均为典型的线性结构。

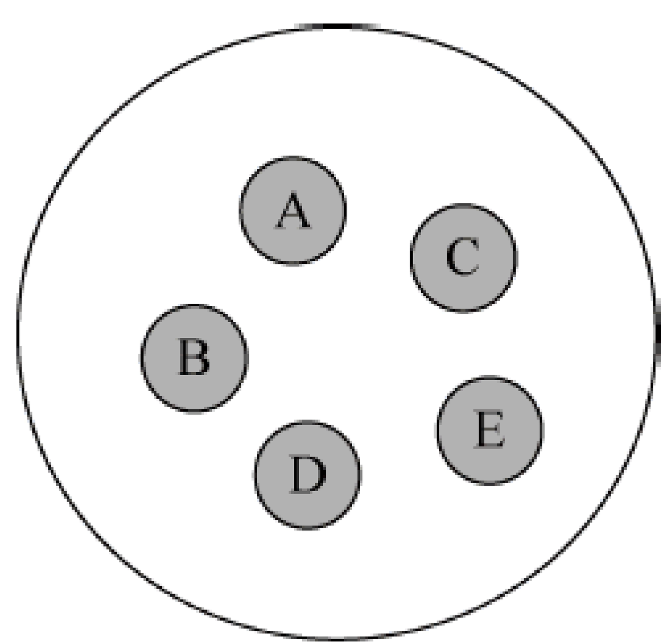


图 1.1 集合结构图示

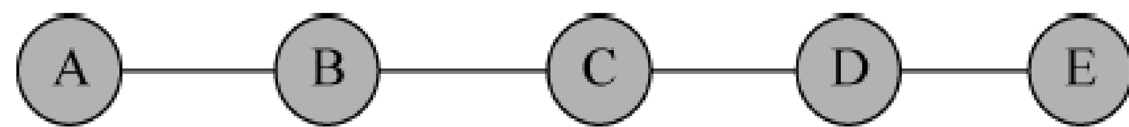


图 1.2 线性结构图示

3. 树形结构

树形结构是指该结构中的结点之间存在一对多的关系。其特点是每个结点最多只有一个前驱，但可以有多个后继，且终端结点可以有多个。从上到下结点间是一对多的关系，从下到上结点间是一对一的关系，二叉树就是一种典型的树形结构。

如图 1.3 所示，树形结构的根结点有且仅有一个（R1 为根结点），每个结点有且只有一个前驱结点（除树根结点外），但可以有多个后继结点（叶结点可看作具有 0 个后继结点）。

图中,R1 的后继结点为 R2 和 R3,R2 的后继结点为 R4 和 R5,R4 和 R5 的前驱结点为 R2,R2 和 R3 的前驱结点为 R1。例如,家族的族谱、公司的人事组织结构等均为典型的树形结构。

4. 图形结构

图形结构是指该结构中的结点之间存在多对多的关系。其特点是:每个结点的前驱和后继的个数都可以是任意的。图形结构可能没有开始结点和终端结点,也可能有多个开始结点、终端结点。

如图 1.4 所示,每个结点有多个前驱结点和多个后继结点。结点北京的后继结点为成都,结点北京的前驱结点为郑州和沈阳,结点郑州的后继结点为北京和沈阳,结点郑州的前驱结点为成都和武汉,其他结点以此类推。例如,青岛市各建筑物的分布图、公交换乘线路等均为典型的图形结构。

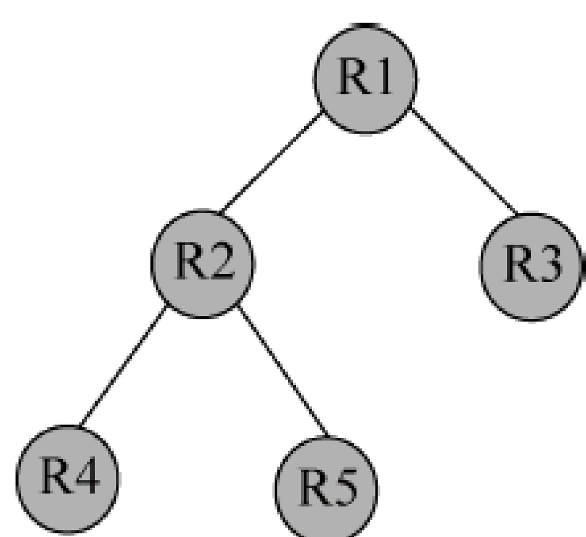


图 1.3　树形结构图示

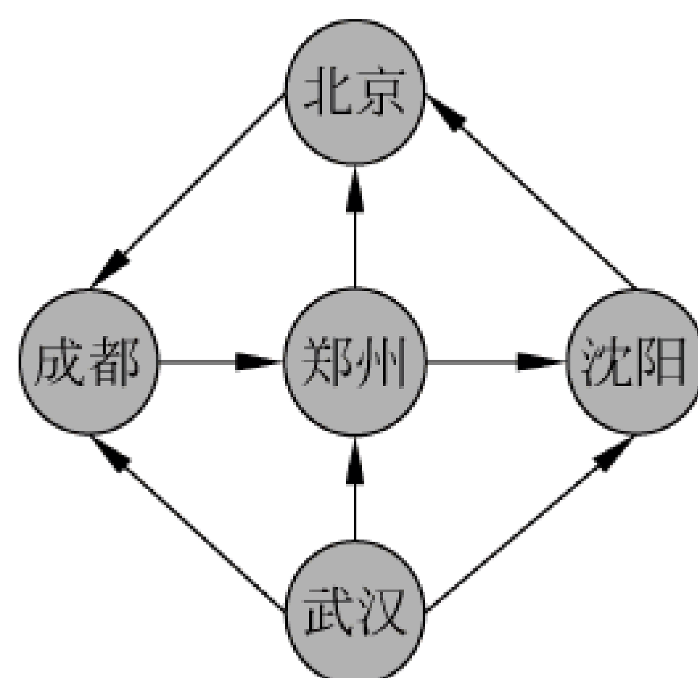


图 1.4　图形结构图示

注意:树形结构和图形结构统称为非线性结构,该类结构中的结点之间存在一对多或多对多的关系。由图形结构、树形结构和线性结构的定义可知,线性结构是树形结构的特殊情况,而树形结构又是图形结构的特殊情况。

【例 1.1】　有一种数据结构 B1=(D,R),其中:

```
D={a,b,c,d,e}
R={<a,b>,<b,c>,<c,d>,<d,e>,<e,a>}
```

画出其逻辑结构表示。

解:B1 对应的逻辑结构图如图 1.5 所示。

从该例中可以看出,每个结点都有一个前驱结点和一个后继结点。这种数据结构的特点是:每个结点的前驱和后继的个数都是任意的,数据元素之间是多对多的关系,即图形结构。

【例 1.2】　有一种数据结构 B2=(D,R),其中:

```
D={a,b,c,d,e}
R={<a,b>,<b,c>,<c,d>}
```

画出其逻辑结构表示。

解:B2 对应的逻辑结构图如图 1.6 所示。

从该例中可以看出,结点 a、b、c、d 之间满足线性结构关系,而结点 e 和其他结点不存在逻辑关系,可以认为,结点 e 和其他结点的关系是任意的,于是整个数据结构的特点满足数据元素之间是多对多的关系,即图形结构。

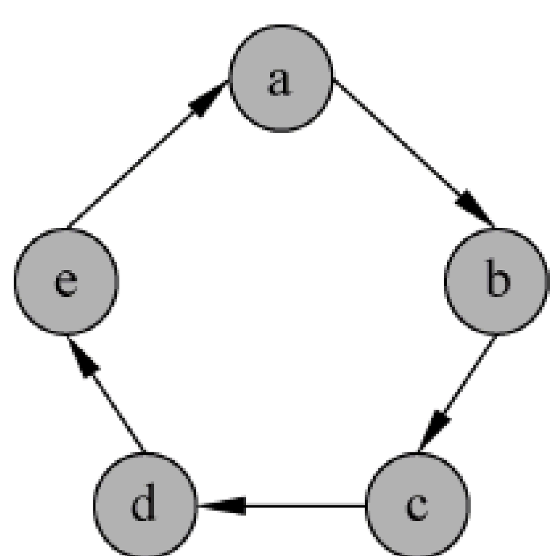


图 1.5　B1 对应的逻辑结构图

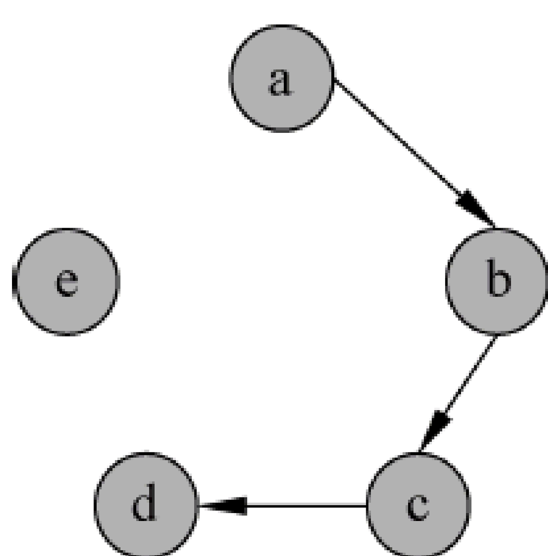


图 1.6　B2 对应的逻辑结构图

【例 1.3】 有一种数据结构 B3＝(D,R),其中:

```
D={a,b,c,d,e}
R={<a,b>,<b,c>,<c,d>,<d,e>}
```

画出其逻辑结构表示。

解:B3 对应的逻辑结构图如图 1.7 所示。

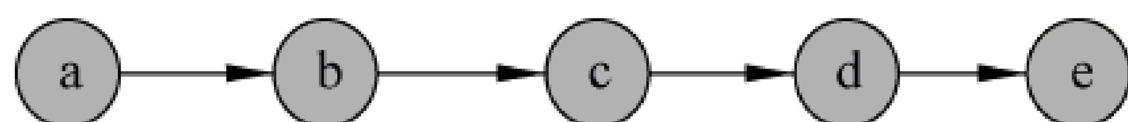


图 1.7　B3 对应的逻辑结构图

从该例中可以看出,每个数据结点有且仅有一个前驱结点(除第一个结点外),有且仅有一个后继结点(除最后一个结点外)。这种数据结构的特点是结点之间为一对一的关系,即线性结构。

【例 1.4】 已知矩阵 $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 6 & 3 \end{bmatrix}_{2\times 3}$,使用二元组形式表示其数据结构。



视频讲解

解:B＝(D,R)

```
D={(1,1,1),(1,2,3),(1,3,5),(2,1,2),(2,2,6),(2,3,3)}
R={r1,r2}
r1={<(1,1,1),(1,2,3)>,<(1,2,3),(1,3,5)>,<(2,1,2),(2,2,6)>,<(2,2,6),(2,3,3)>}
r2={<(1,1,1),(2,1,2)>,<(1,2,3),(2,2,6)>,<(1,3,5),(2,3,3)>}
```

1.1.4　数据的存储结构



视频讲解

研究数据结构的目的是在计算机中对其操作,为此还需要研究如何在计算机中表示数据结构。数据结构在计算机中的映像称为数据的**物理结构**,或称**存储结构**。它所研究的是数据结构在计算机中的实现方法,包括数据结构中元素的表示以及元素间关系的表示。常见的存储结构有顺序存储、链式存储、索引存储,以及哈希(或散列)存储。显然,物理结构不同于逻辑结构,它依赖于计算机,是具体的。

1. 顺序存储结构

顺序存储结构是把逻辑上相邻的结点存储在物理位置上相邻的存储单元里,结点之间的逻辑关系由存储单元的邻接关系体现。通常,顺序存储结构是借助计算机程序设计语言(如 C/C++/Java)的数组描述的。

顺序存储结构的特点是：逻辑上相邻的元素，其物理位置必定相邻。于是，顺序存储元素必在一片连续的存储空间中。

若将例 1.3 的数据结构 B3 采用顺序存储结构存储，则其物理结构如图 1.8 所示。

内存地址	
200	a
201	b
202	c
203	d
204	e

图 1.8　例 1.3 对应的顺序存储结构

顺序存储是把逻辑上相邻的元素存储在物理位置相邻的存储单元中。它的主要优点是节省存储空间，因为分配给数据的存储单元全部用于存放结点的数据(不考虑 C/C++/Java 语言中数组需要指定最大限值的情况)，结点之间的逻辑关系没有占用额外的存储空间，因此存储密度大，空间利用率高，元素存放便于集中管理。可以通过计算直接确定数据结构中的任意一个结点的存储地址，便于随机读取元素，但是插入和删除都会引起大量的结点移动。

2. 链式存储结构

链式存储结构不要求逻辑上相邻的结点在物理位置上也相邻，结点间的逻辑关系是由附加的指针表示的。通常，链式存储结构需借助计算机程序设计语言(如 C/C++)的指针类型描述。

链式存储结构的特点是：逻辑上相邻的元素，其物理位置不一定相邻。

链式存储方法的主要优点是便于修改，在进行插入、删除运算时，仅需要修改相应结点的指针域，不必移动结点。但与顺序存储方法相比，链式存储方法的主要缺点是存储空间的利用率低，因为分配给数据的存储单元有一部分被用来存储结点间的逻辑关系。另外，由于逻辑上相邻的结点在存储空间中不一定相邻，所以不能对结点进行随机存取。

若将例 1.3 的数据结构 B3 采用链式存储结构存储，则其物理结构如图 1.9 所示。

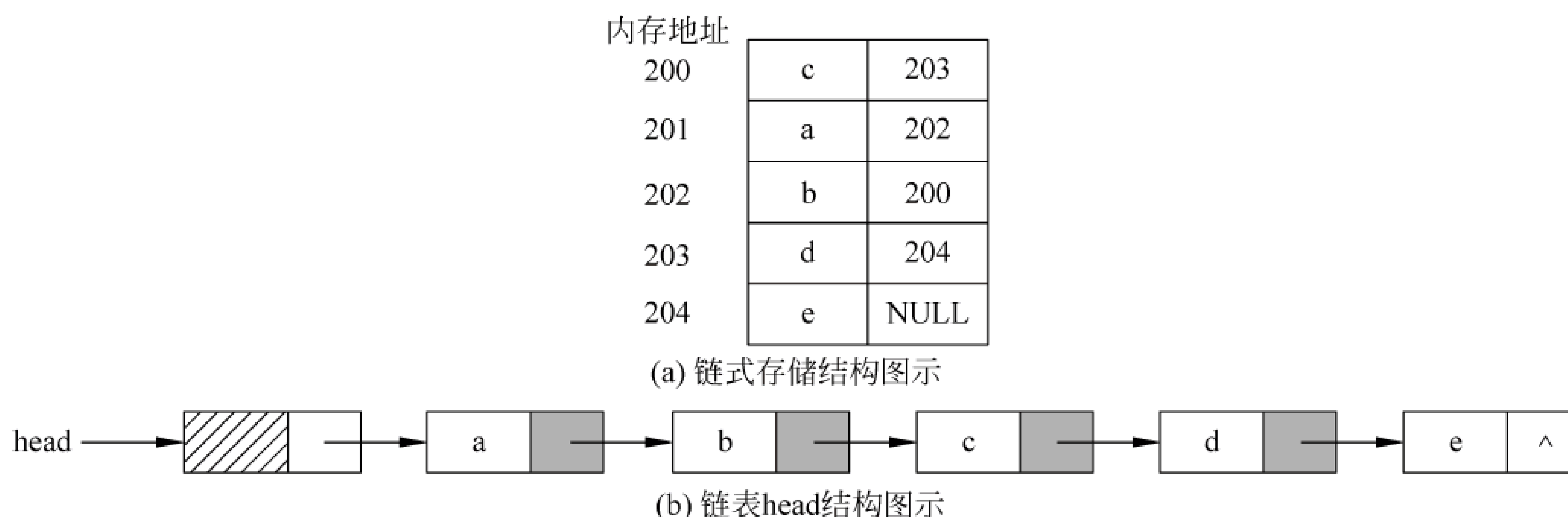


图 1.9　例 1.3 对应的链式存储结构

如图 1.9(a)所示，链式存储方法不仅存储结点的值，而且还存储结点间的关系，即结点由两部分组成：一是存储数据本身的值；二是存储该结点后继结点的地址。特别地，最后一个结点无后继结点，因此其后继结点的地址为 NULL(空)。链式存储结构的表示往往简化如图 1.9(b)所示。

很明显，链式存储结构比顺序存储结构存储密度小，存储空间利用率低。删除和插入操作灵活，不必移动结点，只要改变结点中指针域的值即可。

3. 索引存储结构

索引存储结构通常是在存储结点信息的同时，还建立附加的索引表。索引表中的每一

项称为索引项,索引项的一般形式是:(最大关键字的值,块内首地址)。关键字能唯一确定一个结点。这种带有索引表的存储结构可以大大提高数据查找的速度。

假设一个线性表采用顺序表R存储,其中包含25个元素,其关键字序列为(6,15,10,8,9,24,35,20,28,31,45,50,39,65,75,80,85,96,90,78,100,106,120,99,116)。假设将25个元素分为5块(b=5),每块中有5个元素(s=5),则该线性表的索引存储结构如图1.10所示。由图1.10可见,0～4序号之间代表第一个块,在第一块中最大的关键字为15,5～9序号之间代表第二个块,在第二块中最小的关键字为20,形成第一块中最大的关键字小于第二块中最小的关键字。第二块中最大的关键字为35,10～14序号之间代表第三个块,第三块中最小的关键字是39,形成了第二块中最大的关键字小于第三块中最小的关键字,其他块之间以此类推。从总体关系可以看出块内无序,块间有序的存储结构。

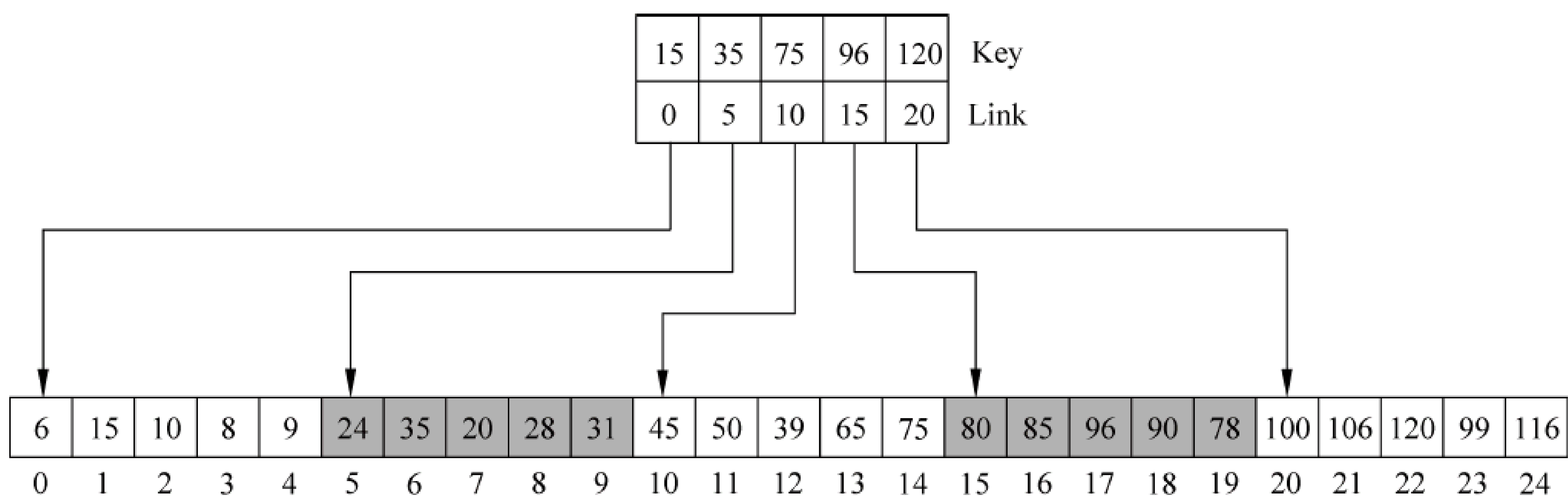


图1.10　索引存储结构图示

线性结构采用索引存储方法后,可以对结点进行随机访问。在进行插入、删除运算时,只移动索引表中对应结点的存储地址,而不必移动结点表中的数据,所以表现出较高的数据修改运算效率。索引存储方法的缺点是增加了索引表,从而降低了存储空间的利用率。

4. 散列(或哈希)存储结构

散列存储结构的基本思想是根据结点的关键字通过哈希(或散列)函数直接计算出一个值,并将这个值作为该结点的存储地址。

哈希存储即寻找一个哈希函数,将不同的关键字根据哈希函数映射出不同的哈希地址,从而实现存储的方法。如图1.11所示,一旦建立了哈希表,在哈希表中进行查找的方法就是以查找关键字K为映射函数的自变量、以建立哈希表时使用的同样的哈希函数(H(K))为映射函数得到一个哈希地址(该地址中原对象的关键字为K_i),将K_i与K进行关键字比较,如果$K_i=K$,则查找成功;否则,以建立哈希表时使用的同样的哈希冲突函数得到新的哈希地址(设该地址中对象的关键字为K_j),将K_j与K进行关键字比较,如果$K_j=K$,则查找成功,否则以同样的方式继续查找,直到查找成功或查找完m个存储单元仍未查找到(即查找失败)为止。

下标	0	1	2	3	4	5	6	7	8	9	10	11
K	77		55	15	42	32	28	45	59	75	88	

图1.11　散列(或哈希)存储结构图

上述4种基本的存储方法,既可以单独使用,也可以组合起来对数据结构进行存储映像。同一种逻辑结构采用不同的存储方法,可以得到不同的存储结构,选择何种存储结构表示相应的逻辑结构,应视具体要求而定,主要考虑运算是否便于计算算法的时间、空间效率。

1.2 抽象数据

1.2.1 数据类型

数据类型是和数据结构密切相关的一个概念,两者容易混淆。在高级程序设计语言中,每个变量、常量或表达式都有一个它所属的确定的数据类型。类型明确或隐含地规定了在程序执行期间变量或表达式所有可能的取值范围,以及在这些值上允许进行的操作。因此,数据类型是一组性质相同的值的集合和定义在此集合上的一组操作的总称。

例如,在高级语言中已实现的,或非高级语言直接支持的数据结构即为数据类型。在程序设计语言中,一个变量的数据类型不仅规定了这个变量的取值范围,而且定义了这个变量可用的运算,如C语言中定义变量i为int类型,则它的取值为−32 768～32 767(16位系统),可用的运算有+、−、*、/、%等。

总之,数据结构是指计算机处理的数据元素的组织形式和相互关系,而数据类型是某种程序设计语言中已实现的数据结构。在程序设计语言提供的数据类型支持下,就可以根据从问题中抽象出来的各种数据模型,逐步构造出描述这些数据模型的新的数据结构。

下面总结C/C++语言中常用的数据类型。

1. C/C++语言的基本数据类型

C/C++语言中的基本数据类型有int型、bool(布尔)型、float型、double型和char型。

2. C/C++语言的指针类型

C/C++语言允许直接对存放变量的地址进行操作。如定义了char i,则&i表示变量i的地址,也称作指向变量i的指针。存放地址的变量称作指针变量。

有关指针的两个操作是:定义了int a,则&a操作是取变量a的地址;定义了int *p(这里的p是指向一个整数的指针),则*p操作是取p指针的值,即取p所指地址的内容。

3. C/C++语言的数组类型

数组是同一类型的一组有序数据元素的集合。数组有一维数组和多维数组。数组名标识一个数组,下标指示一个数组元素在该数组中的顺序位置。数组下标的最小值称为下界,在C/C++中,数组下界从0开始。数组下标的最大值称为上界,在C/C++中,数组上界为数组的大小减1。例如,char a[10]定义了包含10个字符型的数组a,这10个数组元素为a[0],a[1],a[2],…,a[9]。

4. C/C++语言中的结构体类型

结构体由一组称作结构体成员的项组成,每个结构体成员都有自己的标识符。例如:

```
struct student {
int no;              //学号
char name[8];        //姓名
int age;             //年龄
};
```

定义了一个结构体类型 student。下面的语句定义了该类型的两个变量 s1 和 s2。

```
struct student s1, s2;
```

5. C/C++语言中的共用体类型

共用体是把不同的成员组织为一个整体,在存储器中共享一段存储单元,但不同的成员以不同的方式被解释。例如:

```
union data{
int a;
char b;
};
```

定义了一个共用体类型 data。下面的语句定义了该类型的一个变量 t。

```
union data t;
```

变量 t 的整型成员 a 和字符数组成员 b 共享相同的存储单元。

6. C/C++语言中的自定义类型

C/C++语言中允许使用 typedef 关键字指定一个新的数据类型名,例如:

```
typedef int elemtype;
```

将 int 类型与 elemtype 标识符等同起来,这样做的目的是提高程序的可读性。

7. 引用运算符

C++语言中提供了一种引用运算符"&",当建立引用时,程序用另一个已定义的变量或对象(目标)的名字初始化它,从那时起,引用就作为目标的别名使用,对引用的改动实际就是对目标的改动。例如:

```
int a=4;
int &b=a;
```

第 2 个语句声明变量 b 是变量 a 的引用,b 等于 4,之后这两个变量同步改变。

引用常用于函数形参中,采用引用型形参时,在函数调用时将形参的改变回传给实参。例如,有如下函数(其中的形参均为引用型形参):

```
void swap(int &x,int &y)
{
    int temp=x;
    x=y;
    y=temp;
}
```

当执行语句 swap(a,b)时,实参 a 和 b 的值发生交换。如果 swap 函数的形参不用引用类型,这样调用时,由于 C/C++采用实参到形参的单向值传递,所以实参 a 和 b 的值并不发

生任何改变。

1.2.2 抽象数据类型的表示与实现

1. 抽象数据类型

抽象数据类型(Abstract Data Type,ADT)指的是用户进行软件系统设计时从问题的数学模型中抽象出来的逻辑数据结构和逻辑数据结构上的运算,不考虑计算机的具体存储结构和运算的具体实现算法。抽象数据类型中的数据对象和数据运算的声明与数据对象的表示和数据运算的实现相互分离。

抽象数据类型和数据类型实质上是一个概念。例如,各个计算机都拥有的“整数”类型是一个抽象数据类型,尽管它们在不同处理器上实现的方法可以不同,但由于其定义的数学特性相同,在用户看来都是相同的。因此,“抽象”的意义在于数据类型的数学抽象特性。

另一方面,抽象数据类型的范畴更广,它不再局限于前述各处理器中已定义并实现的数据类型(也可称这类数据类型为固有数据类型),还包括用户在设计软件系统时自己定义的数据类型。为了提高软件的复用率,近代程序设计方法学中指出,一个软件系统的框架应建立在数据之上,而不是建立在操作之上(后者是传统的软件设计方法所为),即在构成软件系统的每个相对独立的模块上,定义一组数据和施于这些数据上的一组操作,并在模块内部给出这些数据的表示及其操作的细节,而在模块外部使用的只是抽象的数据和抽象的操作。显然,所定义的数据类型的抽象层次越高,含有该抽象数据类型的软件模块的复用程度也就越高。

一个包含抽象数据类型的软件模块通常应包含定义、表示和实现 3 个部分。

如前所述,抽象数据类型的定义由一个值域和定义在该值域上的一组操作组成。若按其值的不同特性,可细分为下列 3 种类型。

1) 原子类型

原子类型(atomic data type)的变量的值是不可分解的。这类抽象数据类型较少,因为一般情况下,已有的固有数据类型足以满足需求,但有时也有必要定义新的原子数据类型,如数位为 10 的整数。

2) 固定聚合类型

固定聚合类型(fixed-aggregate data type)的变量,其值由确定数目的成分按某种结构组成。例如,复数是由两个实数依确定的次序关系构成。

3) 可变聚合类型

可变聚合类型(variable-aggregate data type)和固定聚合类型相比,构成可变聚合类型“值”的成分的数目不确定。例如,可定义一个“有序整数序列”的抽象数据类型,其中序列长度是可变的。

显然,后两种类型可统称为结构类型。

2. 抽象数据类型的定义

和数据结构的形式定义对应,抽象数据类型可用以下三元组表示。

```
ADT=(D,S,P)
```

其中，D 是数据对象，S 是 D 上的关系集，P 是对 D 的基本操作集。本书采用以下格式定义抽象数据类型。

```
ADT 抽象数据类型名{
数据对象:(数据对象的定义)
数据关系:(数据关系的定义)
基本操作:(基本操作的定义)
}ADT 抽象数据类型名
```

其中，数据对象和数据关系的定义用伪码描述，基本操作的定义格式为

```
基本操作名:(参数表)
  初始条件:(初始条件描述)
  操作结果:(操作结果描述)
```

基本操作有两种参数：赋值参数只为操作提供输入值；引用参数以 & 打头，除可提供输入值外，还将返回操作结果。“初始条件”描述了操作执行之前数据结构和参数应满足的条件，若不满足，则操作失败，并返回相应的出错信息。“操作结果”说明了操作正常完成之后，数据结构的变化状况和应返回的结果。若初始条件为空，则省略之。

【例 1.5】 抽象数据类型三元组的定义。

```
ADT Triplet{
数据对象:D= {e1,e2,e3|e1,e2,e3∈ElemSet(定义了关系运算的某个集合)}
数据关系:R={<e1,e2>,<e2,e3>}
基本操作:
InitTriplet(&T, v1,v2,v3)
    操作结果:构造了三元组 T,元素 e1,e2 和 e3 分别被赋予参数 v1、v2 和 v3 的值。
DestroyTriplet(&T)
    初始条件:三元组 T 已存在。
    操作结果:三元组 T 被销毁。
Get(T, i, &e)
    初始条件:三元组 T 已存在。
    操作结果:用 e 返回 T 的第 i 元的值,1≤i≤3。
Put(&T,i, e)
    初始条件:三元组 T 已存在。
    操作结果:改变 T 的第 i 元的值为 e,1≤i≤3。
IsAscending(T)
    初始条件:三元组 T 已存在。
    操作结果:如果 T 的 3 个元素按升序排列,则返回 1,否则返回 0。
IsDescending(T)
    初始条件:三元组 T 已存在。
    操作结果:如果 T 的 3 个元素按降序排列,则返回 1,否则返回 0。
Max(T,&e)
    初始条件:三元组 T 已存在。
    操作结果:用 e 返回 T 的 3 个元素中的最大值。
Min(T,&e)
    初始条件:三元组 T 已存在。
    操作结果:用 e 返回 T 的 3 个元素中的最小值。
}ADT Triplet
```

多形数据类型(polymorphic data type)是指其值的成分不确定的数据类型。例如,例1.5中定义的抽象数据类型Triplet,其元素e_1,e_2和e_3可以是整数或字符或字符串,甚至由多种成分构成(只要能进行关系运算即可)。然而,不论其元素具有何种特性,元素之间的关系相同,基本操作也相同。从抽象数据类型的角度看,由于其具有相同的数学抽象特性,故称之为多形数据类型。显然,须借助面向对象的程序设计语言(如C++等)实现之。本书中讨论的各种数据类型大多是多形数据类型,限于采用类C/C++语言作为描述工具,故只讨论含有确定成分的数据元素的情况。如例1.5中的ElemSet是某个确定的、将由用户自行定义的、含某个关系运算的数据对象。

1.3 算法及其性能分析



视频讲解

1.3.1 算法

算法(**algorithm**)是对特定问题求解步骤的一种描述,它是指令的有限序列,其中每一条指令表示一个或多个操作。一个算法具有下列5个重要特性。

1. 有穷性

一个算法必须总是(对任何合法的输入值)在执行有穷步之后结束,且每一步都可在有穷时间内完成。

2. 确定性

算法中的每一条指令必须有确切的含义,以便读者理解时不会产生二义性。并且,在任何条件下,算法只有唯一的一条执行路径,即对于相同的输入,只能得出相同的输出。

3. 可行性

一个算法是可行的,即算法中描述的操作都是可以通过执行有限次已经实现的基本运算实现的。

4. 输入

一个算法有**零个**或**多个**输入,这些输入取自于某个特定的对象的集合。

5. 输出

一个算法有**一个**或**多个**输出,这些输出是同输入有某些特定关系的量。

算法的含义和程序十分相似,但是本质却是不同的。例如,一个程序并不需要满足上述的第一个特点。例如,操作系统程序,只要整个系统不遭受破坏,操作系统程序就永不结束。此外,程序是使用机器可执行的语言书写的,而算法通常没有这种限制。算法的描述可以采用文字叙述,也可以采用传统的流程图、N-S图或者PAD图等,本书采用类C/C++语言伪代码描述。

1.3.2 算法设计的目标

算法设计应满足以下4个目标。

1. 正确性

算法应当满足具体问题的需求。通常,一个大型问题的需求应以特定的规格说明方式给出,而一个实际问题或练习题往往就不那么严格,目前多数是用自然语言描述需求,它至

少应当包括对于输入、输出和加工处理等的明确的无歧义性的描述。设计或选择的算法应当能正确地反映这种需求；否则，算法的正确与否的衡量准则就不存在了。

假设有如下程序片段：输入 3 个整数 a、b、c，分别作为三角形的三条边，通过程序判断这 3 条边是否能构成三角形？如果能构成三角形，则判断三角形的类型(等边三角形、等腰三角形、一般三角形)。要求输入 3 个整数 a、b、c，必须满足以下条件：1≤a≤200；1≤b≤200；1≤c≤200。

```
void IsTriangle (int a, int b, int c)
{
    if ((a+b<=c) || (a+c<=b) || (b+c<=a))
    { printf("不能构成三角形"); }
    else
    { if ((a==b) || (b==c) || (a==c))
      { if ((a==b) && (b==c))
        { printf("等边三角形"); }
       else
        { printf("等腰三角形"); }
       }
      else
       { printf("一般三角形"); }
     }
}
```

“正确”一词的含义在通常的用法中有很大差别，大体可分为以下 4 个层次。

- 程序不含语法错误，上述程序片段使用 C 语言完成编写，无语法错误。
- 程序对于几组输入数据，能够得出满足规格说明要求的结果；上述程序中，如果输入 a=3，b=2，c=4，就输出“一般三角形”；如果输入 a=2，b=2，c=2，就输出“等边三角形”；如果输入 a=2，b=2，c=3，就输出“等腰三角形”；如果输入 a=1，b=2，c=1，就输出“不能构成三角形”。
- 程序对于精心选择的典型、苛刻而带有刁难性的几组输入数据，能够得出满足规格说明要求的结果；针对三角形的程序，规定 a，b，c 的输入为 1≤a，b，c≤200。采用软件测试中的边界值测试方法，可以选取边界值 1，2，100，199，200 等几个特殊的数进行测试，该程序均能满足规格说明要求的结果。
- 程序对于一切合法的输入数据，都能产生满足规格说明要求的结果。显然，达到第 d 层意义下的正确是极为困难的，所有不同输入数据的数量大得惊人，逐一验证的方法是不现实的。对于大型软件，需要进行专业测试，而一般情况下，通常以第 3 层意义的正确性(correctness)作为衡量一个程序是否合格的标准。

2. 可读性

算法主要是为了便于人们阅读与交流，其次才是机器执行。好的可读性(readability)有助于人对算法的理解；晦涩难懂的程序易于隐藏较多错误，难以调试和修改。

一些读者喜欢在每行代码前面写一句注释，例如：

```
//成员列表的长度>0 并且 0 && memberList.size() < 200) {
//返回当前成员列表
return memberList;
}
```

看起来似乎很好懂,但是几年之后,这段代码就变成了:

```
//成员列表的长度>0 并且 0 && memberList.size() < 200 || (tmp.isOpen() && flag)) {
//返回当前成员列表
return memberList;
}
```

再之后可能会改成这样:

```
//成员列表的长度>0 并且 0 && memberList.size() < 200 || (tmp.isOpen() && flag)) {
//返回当前成员列表
// return memberList;
//}
if(tmp.isOpen() && flag) {
return memberList;
}
```

随着项目的推进,无用的信息会越积越多,最终甚至让人无法分辨哪些信息是有效的,哪些是无效的。以上代码的书写可读性就比较差。

3. 健壮性

当输入数据非法时,算法也能适当地做出反应或进行处理,而不会产生莫明其妙的输出结果。例如,在求三角形的程序中,假设输入的数据为 a=3,b=3,c=0,输出的结果为"非三角形",但实际是三角形的第三边 c=0 为非法的输入数据,程序没有对非法的数据做出反应或处理。

4. 效率与低存储量需求

通俗地说,效率指的是算法执行的时间。对于同一个问题,如果有多个算法可以解决,执行时间短的算法效率高。存储量需求指算法执行过程中所需要的最大存储空间。效率与低存储量需求都与问题的规模有关。求 1+2+…+100 的和与求 1+2+…+10000 的和的执行时间或运行空间显然有一定的差别。

解决同一个问题总是存在着多种算法,每种算法都要对算法的执行时间(即时间复杂度)和所使用的空间资源(即空间复杂度)进行分析。

时间复杂度:算法执行所需的总时间。

空间复杂度:算法所需的额外空间开销。


视频讲解

1.3.3 算法的时间复杂度度量

算法的时间复杂度又称计算复杂度。算法执行时间需通过依据该算法编制的程序在计算机上运行时所消耗的时间度量。度量一个程序的执行时间通常有两种方法。

1. 事后统计的方法

因为很多计算机内部都有计时功能，有的甚至可精确到毫秒级，不同算法的程序可通过一组或若干组相同的统计数据以分辨优劣。但这种方法有两个缺陷：一是必须先运行依据算法编制的程序；二是所得时间的统计量依赖于计算机的硬件、软件等环境因素，有时容易掩盖算法本身的优劣。因此，人们常采用另一种事前分析估算的方法。

2. 事前分析估算的方法

一个用高级程序语言编写的程序在计算机上运行时所消耗的时间取决于下列因素：

- 依据的算法选用何种策略。
- 问题的规模，例如，求 1 到 100 的整数和。
- 书写程序的语言，对于同一个算法，实现语言的级别越高，执行效率越低。
- 编译程序所产生的机器代码的质量。
- 机器执行指令的速度。

显然，同一个算法用不同的语言实现，或者用不同的编译程序进行编译，或者在不同的计算机上运行时，效率均不相同。这表明使用绝对的时间单位衡量算法的效率是不合适的。撇开这些与计算机硬件、软件有关的因素，可以认为一个特定算法"运行工作量"的大小，只依赖于问题的规模（通常用整数量 n 表示），或者说它是问题规模的函数。

一个算法是由控制结构（顺序、分支和循环 3 种）和原操作（指固有数据类型的操作）构成的，算法时间取决于两者的综合效果。为了便于比较同一问题的不同算法，通常的做法是：从算法中选取一种对于所研究的问题（或算法类型）来说是基本操作的原操作，以该基本操作重复执行的次数作为算法的时间量度。

一般情况下，算法中基本操作重复执行的次数是问题规模的某个函数 f(n)，算法的时间量度记作：

$$T(n)=O(f(n))$$

它表示随问题规模的增大，算法执行时间的增长率和 f(n)的增长率是同一数量级，使用大 O 记号称作算法的渐近时间复杂度（asymptotic time complexity），简称**时间复杂度**。

显然，被称作问题的基本操作的原操作应是其重复执行次数和算法的执行时间成正比的原操作，多数情况下，它是最深层循环内的语句中的原操作，它的执行次数和包含它的语句的频度相同。语句的**频度**（frequency count）指的是该语句重复执行的次数。下面通过案例分析常见算法的时间复杂度的计算过程。



视频讲解

【例 1.6】 求累加的程序。

```
float sum (float a [ ], int n)
{
    float s=0.0;              语句【1】
    for(int i=0;i<n;i++)      语句【2】
        s+=a[i];              语句【3】
    return s;                 语句【4】
}
```

该算法包括4个可执行语句【1】、【2】、【3】和【4】。语句【1】定义变量,执行1次;语句2为循环语句,控制变量i从0增加到n,当i=n时,循环才会终止,故语句【2】的频度是n+1,但它的循环体为语句【3】,需要执行n次;语句【4】为返回语句,执行1次。因此,该算法中所有语句的频度之和为

$$f(n)=1+(n+1)+n+1=2n+3$$

因此,该算法的时间复杂度为O(n)。

【例1.7】 求4×4元素矩阵和的函数。

```
int sum (int num[n][n])
{
        int i,j,r=0;              语句【1】
        for(i=0;i<n;i++)          语句【2】
        for(j=0;j<n;j++)          语句【3】
         r+=num[i][j];            语句【4】
        return r;                 语句【5】
}
```

该算法包括4个可执行语句【1】、【2】、【3】、【4】和【5】。语句【1】定义变量,执行1次;语句【2】为循环语句,控制变量i从0增加到n,当i=n时,循环才会终止,故语句【2】的频度是n+1;语句【3】作为语句【2】循环体内的语句,只执行n次,但语句【3】为循环控制语句,控制变量j从0增加到n,当j=n时,循环才会终止,因此语句【3】本身要执行n+1次,所以语句【3】的频度是n(n+1)。同理,语句【4】的频度为n^2;语句【5】为返回语句,执行1次。因此,该算法中所有语句的频度之和为

$$f(n)=1+(n+1)+n(n+1)+n^2+1=2n^2+2n+3$$

因此,该算法的时间复杂度为$O(n^2)$。

【例1.8】 求两个n阶方阵的乘积C=A×B。

```
#define n 100
void MatrixMultiply (int A[n][n],int B[n][n],int C[n][n])
{   int i, j, k;
for (i=1;i<=n;++i)                              语句【1】
for (j=1;j<=n;++j)                              语句【2】
{ C[i][j]=0;                                    语句【3】
 for (k=1;k<=n,k++)                             语句【4】
     C[i][j]=C[i][j]+A[i][k] * B[k][j];         语句【5】
 }
}
```

该算法包括5个可执行语句【1】、【2】、【3】、【4】和【5】。语句【1】为循环语句,控制变量i从0增加到n,当i=n时,循环才会终止,故语句【1】的频度是n+1;语句【2】作为语句【1】循环体内的语句,只执行n次,但语句【2】为循环控制语句,控制变量j从0增加到n,当j=n时,循环才会终止,因此语句【2】本身要执行n+1次,所以语句【2】的频度是n(n+1)。同理,语句【3】的频度为n^2;语句【4】的频度为$n^2(n+1)$;语句【5】的频度为n^3。因此,该算法中所有语句的频度之和为

$$f(n)=(n+1)+n(n+1)+n^2+n^2(n+1)+n^3=2n^3+3n^2+2n+1$$

因此，该算法的时间复杂度为 $O(n^3)$。

【例 1.9】 使用二分查找法查找元素。

```
int BinarySearch(const ElementType A[], ElementType X, int N)
{
int mid, right, left;
right = 0;
left = N - 1;
while(right <= left){
   mid = (right + left)/2;
   if(A[mid] > X)
      left = mid - 1;
   else if(A[mid] < X)
      right = mid + 1;
   else
      return mid;
}
return -1;
}
```

二分法的关键思想是：假设该数组的长度是 N，那么二分后是 N/2，再二分后是 N/4……直到二分到 1 结束(当然，这属于最坏的情况，即每次找到的那个中间点数都不是要找的)，那么，二分的次数就是基本语句执行的次数，假设次数为 x，$N\times(1/2)^x=1$，则 $x=\log_2 n$。

因此，该算法的时间复杂度为 $O(\log_2 n)$。

【例 1.10】 斐波那契数列算法。

```
Fib(0) = 0
Fib(1) = 1
Fib(n) = Fib(n-1) + Fib(n-2)
int Fibonacci(int n)
{
if (n <= 1)
   return n;
else
   return Fibonacci(n-1) + Fibonacci(n-2);
}
```

这里，给定规模 n，计算 Fib(n)所需的时间为计算 Fib(n-1)的时间和计算 Fib(n-2)的时间的和。

$$T(n<=1) = O(1)$$
$$T(n) = T(n-1) + T(n-2) + O(1)$$

假设求 F(5)的数列，则 F(5)递归调用过程如图 1.12 所示。

每次调用都需要执行两次递归，则

$$F(n)= 2^0+2^1+2^2+\cdots+2^{\log n}=2^{\log(n+1)}-1=2^{n-1}$$

因此，该算法的时间复杂度为 $O(2^n)$。

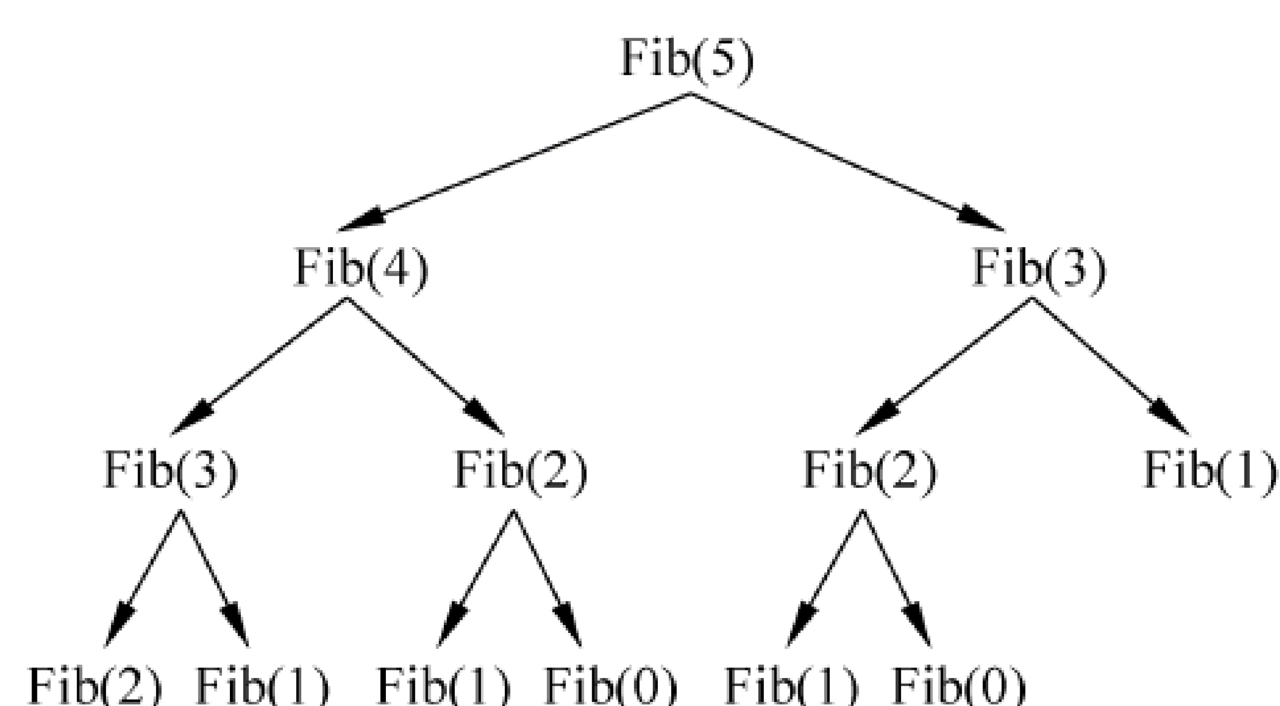


图 1.12 F(5)递归调用过程

【例 1.11】 归并算法。

```
void mergeSort(RcdType SR[ ],RcdType TR[ ], int i, int m, int n)
{
k=i; j=m+1;
while(i<=m && j<=n)
{   if (SR[i].key <= SR[j].key)
      TR[k++]=SR[i++];
   else
      TR[k++]=SR[j++];
}
if (i<=m)
 while(i<=m)
      TR[k++]=SR[i++];
if (j<=n)
 while(j<=n)
      TR[k++]=SR[j++];
}
```

假设给定一个序列 1,2,9,6,8,采用归并算法将数据两两归并,归并步骤如图 1.13 所示。

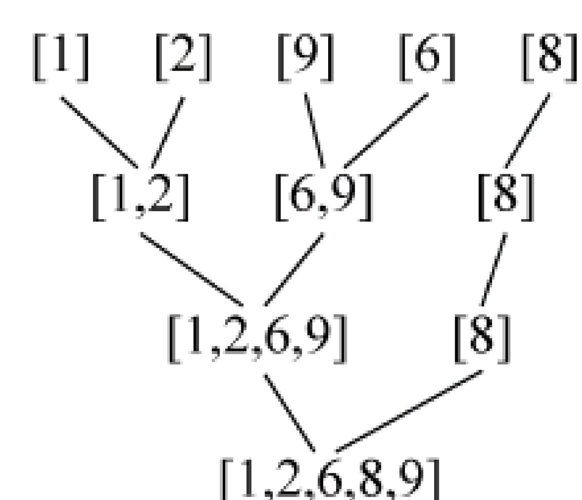


图 1.13 序列归并过程

简单分析一下元素长度为 n 的归并排序所消耗的时间 T(n):调用 mergeSort()函数将给定的序列划分为两部分,每一小部分排序好所花时间为 T(n/2),最后把这两部分有序的数组合并成一个有序的数组,mergeSort()函数所花的时间为 O(n)。

$$T(n)=2T(n/2)+O(n)=O(n\log_2 n)$$

时间复杂度有时与输入有关。一个算法的执行时间 T(n)从理论上无法算出来,必须上机运行测试才能知道,但不可能也没必要对每个算法都测试,只知道哪个算法所需时间更少就可以了。

算法的渐进分析就是要估计当数据规模 n 逐步增大时,T(n)的增长趋势。从数量级大小的比较考虑,当 n 增大到一定值以后,对 T(n)影响最大的就是 n 的幂次最高的项,其他常数项和低幂次项都是可以忽略的。

在各种不同算法中,若算法中的语句执行次数为一个常数,则时间复杂度为 O(1),称为常数阶。常用时间复杂度曲线函数如图 1.14 所示。

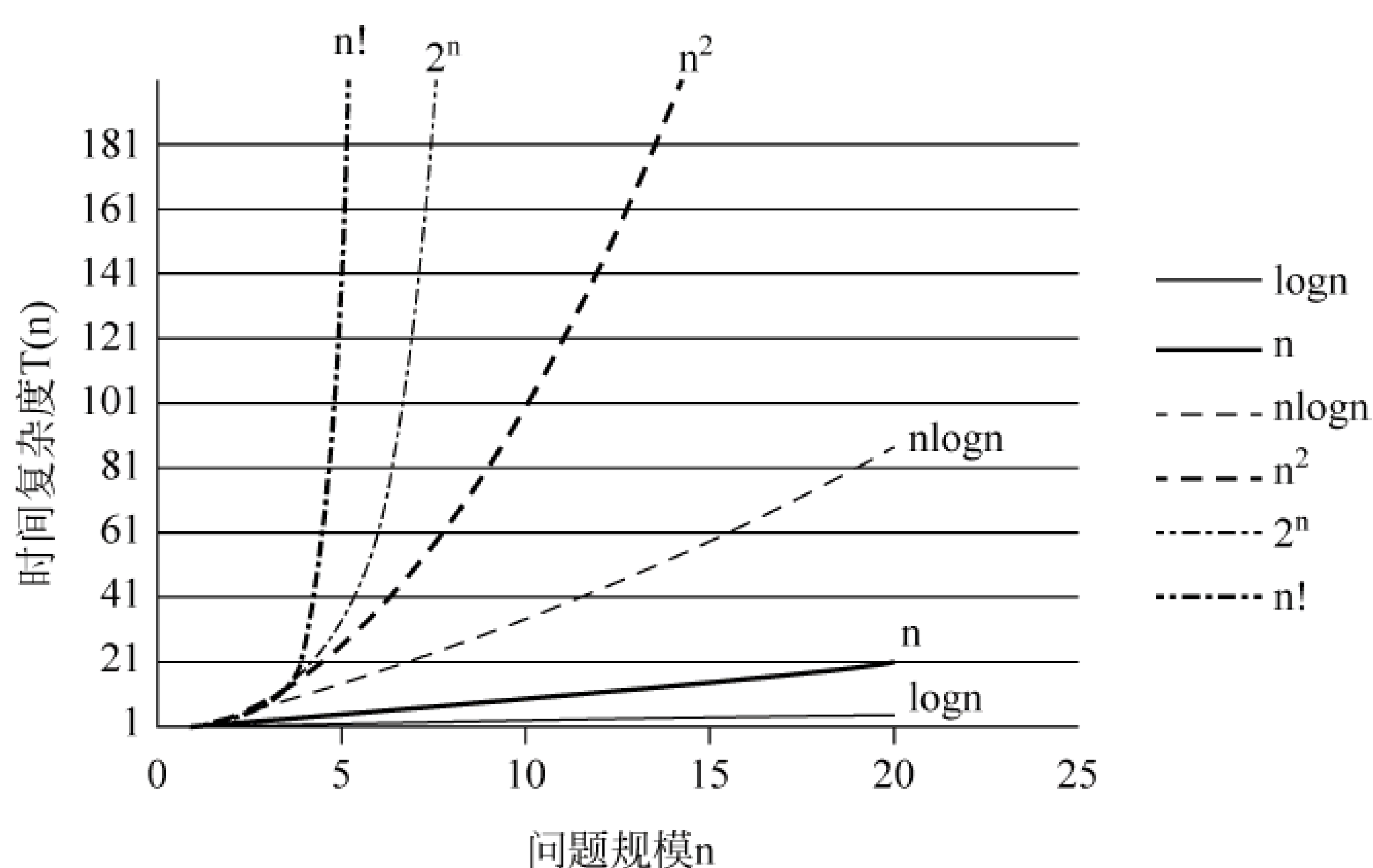


图 1.14 常用时间复杂度曲线函数

一般地，对于足够大的 n，常用的时间复杂性存在以下顺序。

O(1)常数阶＜ $O(\log_2 n)$对数阶＜O(n)线性阶＜$O(n*\log_2 n)$线性对数阶＜$O(n^2)$平方阶＜$O(n^3)$立方阶＜…＜$O(2^n)$指数阶＜O(n!)阶乘阶

一般情况下，对一个问题（或一类算法），只需选择一种基本操作讨论算法的时间复杂度。算法的时间复杂度还可以具体分为最好、最差和平均情况 3 种。在一个算法中，最好的情况下时间复杂度容易计算，但它通常没有太大的实际意义，因为数据具有随机分布性，出现最好情况分布的概率较小；最差情况的时间复杂度也容易求出，它比最好情况有实际意义，通过它可以估计到算法运行时所需要的最长时间，并且提醒用户如何想办法改变数据的排列分布，避免或减少最差情况的发生；平均情况下的时间复杂度的计算要难一些，它往往需要概率统计学方面的知识，但是平均情况最具有实际意义，它能确切地反应运行一个算法的平均快慢程度，通常用平均情况表示一个算法的时间复杂度。一些算法的最差情况和平均情况时间复杂度的数量级是相同的。

1.3.4 算法的空间复杂度度量

一个算法的空间效率是指在算法的执行过程中，所占据的辅助空间数量。辅助空间就是除算法代码本身和输入输出数据所占据的空间外，算法临时开辟的存储空间单元。在有些算法中，占据辅助空间的数量与所处理的数据量有关，而有些却无关。后一种是较理想的情况。在设计算法时，应该注意空间效率。

估计方法是：算法中使用的额外存储单元数量。

与时间复杂度类似，空间复杂度是指算法在计算机内执行时所需存储空间的度量，记作：

$$S(n)=O(f(n))$$

一般讨论的是除正常占用内存开销外的辅助存储单元规模。讨论方法与时间复杂度类似，这里不再赘述。

1.4　STL 概述

1.4.1　STL 的发展和特点

STL(Standard Template Library)是一个具有工业强度的、高效的 C++程序库。它被容纳于 C++标准程序库(C++ Standard Library)中,是 ANSI/ISO C++标准中最新的,也是极具革命性的一部分。该库包含了储存在计算机科学领域里常用的基本数据结构和基本算法,为广大 C++程序员们提供了一个可扩展的应用框架,高度体现了软件的可复用性。有了 STL,不必再从头写太多的标准数据结构和算法,并且可获得非常高的性能。

1.4.2　C++标准库和 STL

1. 容器

容器是存放其他对象的对象。例如常见的 C++内置数组,广义上讲也属于一种容器。容器存放同一种类型的一组元素或对象时,称为同类容器类(homogenous container);存放不同类型的元素或对象时,称为异类容器类(heterogenous container)。STL 容器库包含了两类容器:一类为顺序容器(sequence container);另一类为关联容器(associative container)。容器可用于存放各种类型的数据(基本类型的变量、对象等)的数据结构。

1) 顺序容器

vector:实际上就是一个动态数组。随机存取任何元素都能在常数时间完成。在尾端增删元素具有较佳的性能。

deque:也是一个动态数组,随机存取任何元素都能在常数时间完成(但性能次于 vector)。在两端增删元素具有较佳的性能。

list:STL 实现的 list 由结点组成的双向链表,每个结点都包含一个元素,提供从两个方向遍历元素。双向链表,在任何位置增删元素都能在常数时间完成。list 强大的算法支撑,使其在增加序列元素的同时并不增加读取元素的时间,读取元素的时间仍然为常数,因此 list 支持随机存储。

上述 3 种容器称为顺序容器,是因为元素的插入位置同元素的值无关。

2) 关联容器

set:由结点组成的红黑树,每个结点都包含一个元素,结点之间以某种作用于元素对的谓词排列,没有两个不同的元素能够拥有相同的次序。这个与 map 存储的结构是一致的,都是以二叉树结构的结点形式存放。

multiset:快速查找,可有重复元素。

map:映射提供了一个键/值对,基于键的查询,迅速查找到键相对应的所需的值。如 map<Type1, Type2>**map_name**:其建立的是一个以 Type1 为索引,Type2 的值的查询。

multimap:一对一映射,可有重复元素,基于关键字查找。

上述 4 种容器通常以平衡二叉树方式实现,插入和检索的时间都是 $O(\log_2 n)$。

2. 容器适配器

容器适配器通过修改调整容器的接口,使得容器适用于另一种不同效果。修改顺序容

器接口的容器适配器有 stack 和 queue，其中 stack 是具有后进先出特性的访问受限的线性结构，而 queue 是具有先进先出特性的访问受限的线性结构。此外，还有优先队列。

stack：stack（栈）是一种容器适配器，前面已经讲过，它不是独立的容器，只是某种序列容器的变化，它提供原容器的一个专用的受限接口。默认的 stack 类（定义在<stack>头文件中）是对 deque（双端队列）的一种限制。

queue：队列容器是另外一种容器适配器。它默认通过 deque 实现队列，提供了如 push、pop 等成员函数，还包括测试队列的使用情况、元素个数、是否为空等功能。

priority_queue：优先级高的元素先出。

3. 迭代器

在 C++中，我们经常使用指针，而迭代器就相当于指针，它提供了一种一般化的方法，使得 C++程序能够访问不同数据类型的顺序或者关联容器中的每一个元素，可以称它为“泛型指针”。

STL 定义了 5 种迭代器类型，即前向迭代器（forward iterator）、双向迭代器（bidirectional iterator）、输入迭代器（input iterator）、输出迭代器（output iterator）、随机访问迭代器（random access iterator）。

1）前向迭代器

前向迭代器可用来以一个方向遍历容器的元素，支持容器元素的读写。

2）双向迭代器

双向迭代器可用来从两个方向遍历容器的元素，支持容器元素的读写。

3）输入迭代器

输入迭代器可用来读取容器中的元素，但是不能保证支持向容器写入操作。输入迭代器必须至少支持：两个迭代器的相等和不相等的判断（==，!=）。通过操作符（++）（前置或者后置）使迭代器向前递增指向下一个元素，通过指针操作符（*）完成对元素的读取，还有成员访问操作符（->）。

4）输出迭代器

输出迭代器可以被认为是与输入迭代器相反功能的迭代器。它用来向容器中写入元素，但是不保证支持读取容器的内容。输出迭代器支持的操作至少包括操作符（++）（包括前置和后置）和指针操作符（*）（左值形式）。

5）随机访问迭代器

随机访问迭代器支持容器元素的随机访问，同样支持容器元素的读与写。

1.4.3 数据结构和 STL 的关系

STL 就是建立在模板函数和模板类基础之上的功能强大的库。模板函数可以实现一般化的常用算法（如统计、排序、查找等）。模板类可以实现支持几乎所有类型的容器，用来实现常用的数据结构（如链表、栈、队列、平衡二叉树等）。

【模板例 1】 使用模板求两个数的最大值。

```
template <class T>
T max(T a , T b)
{
```

```
return (a > b) ? a , b;
}
```

template <模板形参表>

```
<返回值类型><函数名>(模板函数形参表)
{
//函数定义体
}
```

【模板例 2】 编写一个对具有 n 个元素的数组 a[]求最小值的程序,要求将求最小值的函数设计成函数模板。

```
#include <iostream>
template <class T>
T min(T a[],int n)
{
    int i;
    T minv=a[0];
    for(i = 1;i < n; i++){
            if(minv>a[i])
        minv=a[i];
    }
    return minv;
}
void main()
{
int a[]={1,3,0,2,7,6,4,5,2};
double b[]={1.2,-3.4,6.8,9,8};
cout<<"a 数组的最小值为:"<<min(a,9)<< endl;
cout<<"b 数组的最小值为:"<<min(b,4)<<endl;
}
```

此程序的运行结果为

```
a 数组的最小值为:0
b 数组的最小值为:-3.4
```

1.5 综合案例

1.5.1 哥德巴赫猜想问题

1. 问题描述

1742 年,德国数学家哥德巴赫给当时住在德国的著名数学家欧拉的一封信中,提出把一个整数表示成素数之和的猜想,这就是著名的“哥德巴赫猜想”,这个猜想表述为下列两个命题。

• 每个大于等于 6 的偶数都是两个奇素数之和。
• 每个大于等于 9 的奇数都可以表示为 3 个奇数之和。

2. 解题思路

哥德巴赫猜想的本质是第一命题，解决了第一命题，第二命题即可迎刃而解。通过数学家们的不懈努力，通过对“哥德巴赫猜想”命题的证明，表明猜想问题的提出是合理的，命题是正确的。下面根据命题一的结论，进行算法设计实现如下。

```
#include <stdio.h>
#include <math.h>
#include <iostream>
using namespace std;
//判断 num 是否为素数
int IsPrimer(int num) {
  int i, pow_num;
  pow_num = sqrt(num);
  for (i = 2; i <= pow_num; i++) {
    if (0 == (num % i))
      return -1;
  }
  return 0;
}
/* 输入：一个大于等于 6 的偶数 max_num
 * 处理：将其拆分成两个奇素数之和
 * 输出：不同的拆分结果
 */
void GoldbachGuess(int max_num) {
  int even,i;
  printf("Goldbach is right:\n");
  //从 2 开始，遍历后续所有偶数
  for(even = 2; even <= max_num; even += 2) {
    for(i = 1; i < even; i++) {
      if(0 == IsPrimer(i) && 0 == IsPrimer(even-i)) {
        printf("%d + %d = %d\n", i, even-i, even);
        break;
      }
    }
  }
}
int main(int argc, char ** argv) {
  int n;
  do {
    cout << "Input a even number." << endl;
    cin >> n;
    GoldbachGuess(n);
    cout << "Continue?(1/0)" << endl;
  } while (cin >> n && 1 == n);
}
```

注意：从素数的定义看，'1'应该是素数，只是能被 1 整除与能自身整除这两个条件重合了，所以'1'是素数，且是奇素数。

1.5.2 连续整数问题

1. 问题描述

连续整数问题属于正整数拆分问题，即将一个正整数 n 分解成若干正整数的和；在不考虑求和顺序的情况下，一般假设 $n=n_1+n_2+\cdots+n_k$，$n_1\geqslant n_2\geqslant\cdots\geqslant n_k$。特别地，连续整数问题是指将正整数 n 表示成两个或者多个连续正整数之和。例如：

$$15=1+2+3+4+5$$

$$15=4+5+6$$

$$15=7+8$$

2. 解题思路

经过数学家们考查论证，有些正整数不能写成连续个正整数的和，如 8。那么，如何判断正整数 n 是否可以写成连续若干个正整数之和呢？若可以进行连续整数的拆分，如何通过算法实现计算整数 n 的所有可能拆分方式的个数(n 的拆分数)呢？

3. 代码实现

```
int Ferrers(int num) {
  int n = 0;
  int i, cnt = 0;
  do
    scanf("%d", &n);
  while (n < 1 || n > 1000);
  for (i = 1; i <= n / 2; i++) {
    int sum = i, j = i + 1;
    while (sum < n)
      sum += j++;
    if (sum == n) {
      cnt++;
    }
  }
  return cnt;
}
```

本章小结

本章主要介绍了数据结构的基本概念，学习要点如下。

- 理解数据结构的定义，数据结构包含的逻辑结构、存储结构和运算三方面的相互关系。
- 掌握各种逻辑结构(即线性结构、树形结构和图形结构)之间的差别。
- 了解各种存储结构(即顺序存储结构、链式存储结构、索引和散列)之间的差别。
- 了解数据结构和数据类型的差别和联系。
- 了解抽象数据类型的概念和说明方式。
- 掌握算法的定义及特性。
- 重点掌握算法的时间复杂度和空间复杂度分析。

第2篇　线性结构篇

第2章　线　性　表

线性结构是最简单也是最常用的一种数据结构，其特点是数据元素之间的逻辑关系是线性关系。线性结构是数据元素间约束力最强的一种数据结构：非空线性结构的有限集合中，存在唯一一个被称为“第一个”的元素；存在唯一一个被称为“最后一个”的元素；除“第一个”元素无前驱外，集合中的每个元素均有且只有一个“直接”前驱（简称前驱）；除“最后一个”元素无后继外，集合中的每个元素均有且只有一个“直接”后继（简称后继）。第2～4章将讨论线性结构。本章介绍线性表的相关概念、线性表的顺序存储结构和链式存储结构以及相关算法的实现。

2.1　线性表的抽象数据类型



视频讲解

2.1.1　线性表的定义

线性表是n个类型相同的数据元素的有限序列。至于每个数据元素的具体含义，在不同的情况下各不相同，它可以是一个数或一个符号，也可以是一页书，甚至其他更复杂的信息。

线性表的例子很多，例如，英文字母表（‘A’，‘B’，…，‘Z’）是一个线性表，表中的每一个英文字母是一个字符元素；再如，医院排队叫号，被叫号码也是一个线性表（1，2，…，100），表中的每一个号码都对应一个整数类型的数据；再如，一个学校的学生信息表见表2.1，表中每个学生的基本信息为一个记录，每个记录按一定次序排列也可以构成一个线性表。

表2.1　学生信息表

姓　名	学　号	性　别	年　龄	班　级	健康状况
王晓	160101	男	18	计应16	良好
程红	160102	女	19	计应16	良好
刘建平	160103	男	18	计应16	一般
张丽丽	160104	女	20	计应16	良好
王亚斌	160105	男	19	计应16	良好

表中每一行是数据元素（又称记录），它由姓名、学号、性别、年龄、班级和健康状况6个数据项（又称字段）组成。“王晓”对应的记录是首记录，“王亚斌”对应的记录是尾记录，“刘建平”的直接前驱是“程红”，其直接后继是“张丽丽”。

综合上述可见，线性表中的数据元素可以是各种各样的，但同一线性表中的元素必定具

有相同的特性,即属同一数据对象,相邻数据元素之间存在着序偶关系。线性表一般表示为

$$L=(a_1,a_2,\cdots,a_{i-1},a_i,a_{i+1},\ \cdots,a_n)$$

其中,L 是线性表的名称;n 是线性表 L 中元素的个数,称为**表长**,当 n=0 时,称为**空表**;a_1 是线性表的第一个元素,称为**首元素**,a_n是线性表的最后一个元素,称为**尾元素**;L 中的第 i 个元素为 a_i,i 称为其**位序**,a_i的前驱是 a_{i-1},a_i的后继是 a_{i+1}。

注意:线性表中不是所有元素都有前驱或都有后继的,如首元素无前驱,尾元素无后继;但若元素有前驱或后继时,有且仅有一个。

线性表 L 对应的数据结构如下所示。

```
L=(D,R)
D={a_i|1≤i≤n,n≥0;a_i 为 elemtype 类型}          //elemtype 是自定义的类型
R={r}
r={<a_i,a_{i+1}>|1≤i≤n-1 }
```

线性表对应的逻辑结构示意图如图 2.1 所示。

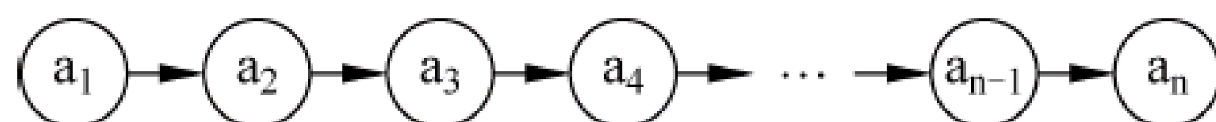


图 2.1　线性表对应的逻辑结构示意图


视频讲解

2.1.2　线性表的抽象数据类型描述

线性表是一个相当灵活的数据结构,它的长度可根据需要增长或缩短,即对线性表的数据元素不仅可以进行随机访问,还可进行插入和删除等。

抽象数据类型线性表的定义如下。

```
ADT List
{
 数据对象:D={a_i|1≤i≤n,n≥0;a_i 为 elemtype 类型}          //elemtype 是自定义的类型
 数据关系:R={r}, r={<a_i,a_{i+1}>| a_i,a_{i+1}∈D, 1≤i≤n-1}
 基本运算:
  InitList(&L):初始化线性表。
      初始条件:线性表 L 不存在。
      操作结果:构造一个空的线性表 L。
  DestroyList(&L):销毁线性表。
      初始条件:线性表 L 存在。
      操作结果:释放线性表 L 占用的内存空间。
  ListEmpty(L):判断线性表是否为空表。
      初始条件:线性表 L 存在。
      操作结果:若 L 为空表,则返回真,否则返回假。
  ListLength(L):计算线性表的长度。
      初始条件:线性表 L 存在。
      操作结果:计算并返回 L 中的元素个数,即表长。
  DispList(L):输出线性表。
      初始条件: 线性表 L 存在。
      操作结果:当线性表 L 非空时,顺序显示 L 中各结点的值域,否则输出表空。
  GetElem(L,i,&e):读取线性表 L 中的某个数据元素值。
```

初始条件:线性表 L 存在且 1≤i≤n。
操作结果:用 e 返回 L 中第 i 个元素的值。
ListInsert(&L,i,e):插入数据元素。
初始条件:线性表 L 存在且 1≤i≤n+1。
操作结果:在 L 中第 i 个位置上插入新的元素 e。
ListDelete(&L,i,&e):删除数据元素。
初始条件:线性表 L 存在且 1≤i≤n。
操作结果:删除 L 的第 i 个位置上的元素,并用 e 返回其值,L 的长度减 1。
LocateElem(L,e):按元素值查找。
初始条件:线性表 L 存在,e 是一个和线性表 L 中元素类型相同的数据元素。
操作结果:返回 L 中第 1 个值域与 e 相等的数据元素的序号,若这样的元素不存在,则返回值为 0。
} ADT List

对线性表还可以进行一些复杂的运算,例如,将两个或多个线性表合并成一个线性表;将一个线性表拆分成两个或多个线性表;复制线性表;将线性表中的元素按某关键字递增或递减重新排列等。用以上基本运算可以实现更为复杂的运算。

【例 2.1】 某线性表 L=(34,89,765,12,90,-34,22),求下面基本运算的执行结果。

解:各种基本运算的结果如下。

```
ListLengh(L)=7              //当前表中有 7 个元素
ListEmpty(L)=false          //当前表中有元素,不为空,返回 false
GetElem(L,3,e)=765          //线性表中第 3 个元素的值为 765
LocateElem(L,12)=4          //元素 12 位于线性表中第 4 个元素的位置
ListInsert(L,4,55)在当前第 4 个元素前插入元素 55,当前第 4 个元素为 12,因此执行该运算后,
线性表 L 变成 L=(34,89,765,55,12,90,-34,22)
ListDelete(L,3)删除线性表中第 3 个位置的元素,执行该运算后,线性表 L 变为
L=(34,89,55,12,90,-34,22)
```

【例 2.2】 假设两个集合 A 和 B,编写一个算法求一个新的集合 C=A∪B。

解:假设使用线性表 La 和 Lb 分别表示集合 A 和 B,于是表中的元素即为集合中的元素,现在通过并集运算产生一个新集合 C,即线性表 Lc。于是,集合的并集运算就成了线性表 La、Lb 合并成线性表 Lc 的运算。假设线性表 La、Lb 中的元素如下。

$$La=(a_1, a_2, a_3, \cdots, a_n)$$
$$Lb=(b_1, b_2, b_3, \cdots, b_m)$$

首先将表 La 中的元素依次取出并插入到表 Lc 中;再依次取出表 Lb 中的元素,若该元素和表 La 中的各个元素均不相同,则可插入表 Lc 中。算法实现如下。

```
void union(List &La, List &Lb, List &Lc)
{
 int La_len, Lb_len, i,j;
 elemtype e;
 La_len= ListLength(La);
 Lb_len= ListLength(Lb);
 for(i=1,j=1; i<=La_len; i++, j++){
   GetElem(La, i, e);
```

```
        ListInsert(Lc,j,e);
      }
    for(i=1; i<=Lb_len; i++){
      GetElem(Lb, i, e);
    if(!LocateElem(Lc, e))
        ListInsert(Lc,++j,e);
    }
  }
```

很明显,该算法的时间复杂度为 O(La_len×Lb_len)或 O(n×m)。

【例 2.3】 已知线性表 LA、LB,表中的数据元素按值非递减有序排列,现要求将 LA 和 LB 归并为一个新的线性表 LC,且 LC 中的数据元素仍按值非递减有序排列。

分析:例如,设

```
LA=(1,3,5,7,8)
LB=(2,6,8,15,20,22)
```

则

```
LC=(1, 2, 3, 5, 6, 7, 8, 15, 20, 22)
```

从上述问题要求可知:LC 中的数据元素或是 LA 中的数据元素,或是 LB 中的数据元素,则先初始化 LC(创建空表 LC),然后将 LA 或 LB 中的元素根据值大小逐个插入 LC 中即可。为使 LC 中的元素按值非递减有序排列,可设两个指针 i 和 j,分别指向 LA 和 LB 中的某个元素,若设 i 当前指的元素为 a_i,j 当前指的元素为 b_j,当前应插入 LC 中的元素为 c_k。显然,指针 i 和 j 的初值均为 1,关于 LA 与 LB 读取的元素比较情况分为以下 3 种。

- 若 $a_i<b_j$,则 $c_k=a_i$,即将 LA 中的元素插入 LC 中,同时 i,k 后移。
- 若 $a_i==b_j$,则 $c_k=a_i$(或 b_j),即将 LA 中的元素(也是 LB 中的元素)插入 LC 中,同时 i,j,k 后移。
- 若 $a_i>b_j$,则 $c_k=b_j$,即将 LB 中的元素插入 LC 中,同时 j,k 后移。

于是,上述过程的算法实现如下。

```
void MergeList(List LA, List LB, List &LC)
{
  //已知线性表 LA 和 LB 中的数据元素按值非递减排列
  //归并 LA 和 LB 得到新的线性表 LC,LC 的数据元素也按值非递减排列
  InitList(LC);
  int i=j=1,k=0;
  La_len=ListLength(LA);Lb_len = ListLength(LB);
  while ((i<= La_ len)&&(j<=Lb_len))          //LA 和 LB 均非空
{   GetElem(LA, i, ai);
    GetElem(LB, j, bj);
    if(ai<=bj)
```

```
    {   ListInsert(LC,++k, ai);
        ++i;
        }
    else
    {   ListInsert(LC,++k, bj);
        ++j;
       }
    }
    while(i<=La_len)
   {   GetElem(LA,i++, ai);
       ListInsert(LC,++k, ai);
     }
     while (j<=Lb_len)
    {   GetElem(LB,j++,bj);
        ListInsert(LC,++k, bj);
    }
}
```

很明显,该算法的时间复杂度为 O(La_len+Lb_len)。

从上面的例子可以看出,算法设计取决于数据逻辑结构,有了逻辑结构,就可以设计算法,然而,算法的实现则依赖数据的存储结构,下面将做出详细说明。

2.2　线性表的顺序存储结构

线性表的顺序存储结构是最简单、最常用的存储方式,它直接将线性表的逻辑结构映射到存储结构上,所以既便于理解,又容易实现。本节讨论顺序存储结构及其基本运算的实现。



视频讲解

2.2.1　线性表的顺序存储结构——顺序表

线性表的顺序存储结构是,把线性表中的所有元素按照其逻辑顺序依次存储到从计算机存储器中指定存储位置开始的一块连续的存储空间中,由此得到的线性表叫顺序表。由于线性表中逻辑上相邻的两个元素在对应的顺序表中它们的存储位置也相邻,所以这种映射称为直接映射。已知线性表 $L=(a_1, a_2, \cdots, a_i, \cdots, a_n)$采用顺序存储,形成的顺序表如图 2.2 所示。

因为内存中的地址空间是线性的,因此,用物理上的相邻实现数据元素之间的逻辑相邻关系简单方便。于是,顺序表在内存中占用一块连续的存储空间,当我们知道了第一个元素 a_1 的地址,也就是指定的存储位置(称为**基地址**,记为 $LOC(a_1)$),假定线性表的元素类型为 elemtype,即每个元素占用的存储空间大小(即字节数)为 sizeof(elemtype),第 $i+1$ 个元素($1\leqslant i\leqslant n$)的存储位置紧接在第 i 个元素的存储位置的后面,以此类推,第 n 个元素存储在下标为 $n-1$ 的位置上。因此,顺序存储方式中只要确定了表的起始位置,表中任意元素都可以随机存取。在 C/C++程序设计语言中的数组类型也有随机存取的特性,因此,通常使用数组表示顺序存储结构。从图 2.2 可发现顺序表中任意元素 a_i的地址 $LOC(a_i)$。

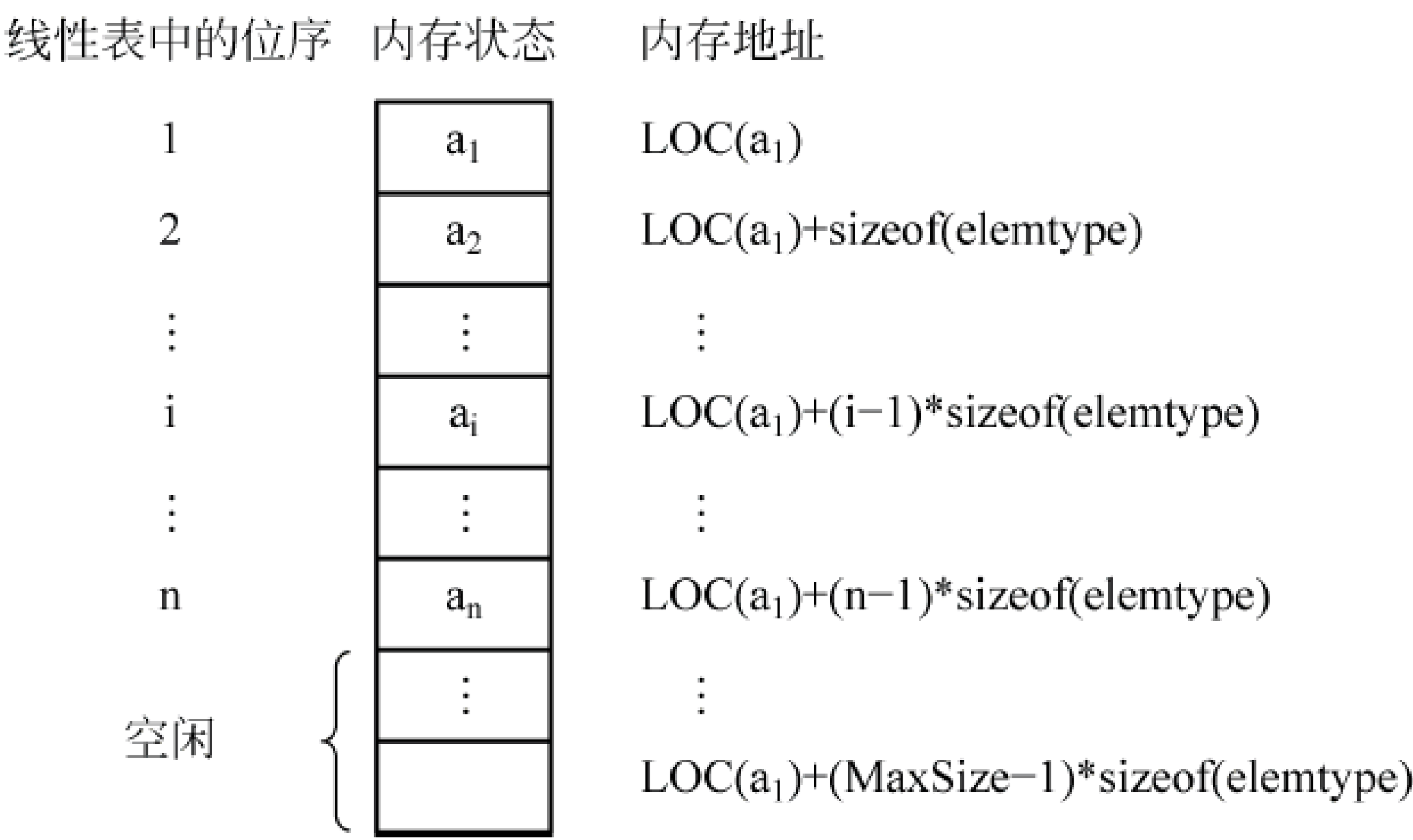


图 2.2　顺序表存储结构示意图

```
LOC(ai)= LOC(a1)+(i-1) * sizeof(elemtype)    1≤i≤n
```

因此,只要确定了顺序表的基地址 LOC(a_1),顺序表中的任意元素 a_i都可以随机存储,所以,顺序表结构是一种随机存取的存储结构。换句话说,以元素在计算机内"物理位置相邻"表示顺序表中数据元素之间的逻辑关系。每一个数据元素的存储位置都与顺序表的起始位置相差一个和数据元素在顺序表中的位序成正比的常数(图 2.2)。

注意:顺序存储结构的特点如下。

- 利用数据元素的存储位置表示线性表中相邻数据元素之间的前后关系,即顺序表的逻辑结构与存储结构(物理结构)一致。
- 访问顺序表时,可以利用上述给出的数学公式快速计算出任何一个数据元素的存储地址。因此可以认为,按位序访问每个数据元素花费的时间相等,均为 O(1)。

【例 2.4】　一维数组 M,下标的范围是 1~9,每个数组元素用相邻的 5 个字节存储。存储器按字节编址,设存储数组元素 M[1]的第一个字节的地址是 98,求 M[3]的第一个字节的地址。

解:地址计算通式为

$$LOC(a_i) = LOC(a_1) + (i-1) * sizeof(elemtype)$$

因此,LOC(M[3]) = 98+(3−1)*5 =108。

假设一个顺序表的元素最大个数用 MaxSize 表示,一般将 MaxSize 定义为一个整型常量。若一个顺序表的元素不会超过 100 个,则可把 MaxSize 定义为 100。

```
#define MaxSize 100
```

顺序存储类型可定义如下。

```
typedef struct
{   elemtype data[MaxSize];          //存放顺序表的数组
    int length;                      //顺序表的长度(即元素个数)
}SqList;
```

上述表述中，顺序表结构被定义为结构体类型的数据，其中 data[ ]数组用来表示顺序表在内存中的存储空间，数组名 data 表示连续存储空间的基地址，MaxSize 表示存储空间的个数，也是最大元素个数，于是，顺序表中的元素就可以被当成 data[ ]数组中的数组元素处理。例如，数组元素可以通过下标读取，此时顺序表中的元素除了通过计算内存地址读取元素以外，还可以使用更简便的下标读取。length 是顺序表中实际元素的个数，则 length≤MaxSize。由于顺序表被定义成静态结构，而实际运算时，随着操作的进行，当 length＝＝MaxSize 时，顺序表已满，无空闲空间，无法实现插入操作，因此，其他一些教材用动态数组表示顺序表。

注意：

- 为了使运算简单，假设 elemtype 为 int 类型，则可使用如下的自定义类型语句。

```
typedef int elemtype;
```

- 顺序表的 data[ ]数组的下标从 0 开始，而顺序表中元素的序号从 1 开始，注意两者之间的转换。
- 既可以采用顺序表指针方式建立和使用顺序表，也可以直接使用顺序表，其定义语句分别为

```
SqList *L;      //顺序表指针 L，即定义指向某已知顺序表的指针 L
SqList L;       //顺序表 L
```

本教材采用顺序表指针的方式进行运算。

例如，对于表 2.1 的逻辑结构学生信息表，假定每个学生信息占用 50 个存储单元，数据从 100 号单元开始由低地址到高地址方向存储，对应的顺序表结构如图 2.3 所示，其中 data[ ]为包含学生姓名、学号、性别、年龄、班级、健康状况的结构体类型的数组，该顺序表的 length 域为 5。

内存地址	
100	王晓
	160101
	男
	18
	计应16
	良好
150	程红
	160102
	女
	19
	计应16
	良好
200	…

图 2.3 表 2.1 对应的顺序表结构

2.2.2 顺序表基本运算的实现

线性表抽象数据类型中定义了线性表的一些基本运算。下面讨论这些运算在顺序存储结构下是如何实现的。

1. 初始化顺序表

顺序表的初始化是构造一个空的顺序表 L。实际上，只分配顺序表的存储空间，并将 length 域设置为 0 即可。

```
void InitList(SqList *&L)
{
L = (SqList *) malloc(sizeof(SqList));
L->length=0;                //将当前线性表长度置 0
 }
```

算法的时间复杂度为 O(1)。

2. 建立顺序表

建立顺序表和初始化顺序表是两种不同的运算,初始化发生的时间靠前,而创建发生的时间靠后,在初始化空的顺序表之后,通过存入给定的若干数据元素,使其由空表到非空表创建起来。在下面的算法中,某个含有 n 个元素的数组 a[ ],将其每个元素依次存放到顺序表 L 中,从而建立顺序表 L,并将 n 赋给顺序表的长度域。算法如下。

```
void CreateList_Sq(SqList *&L,elemtype a[],int n)      //由 a 中的 n 个元素建立顺序表
{   int i;
    L=(SqList *) malloc(sizeof(SqList));                //分配存放线性表的空间
    for(i=0;i<n;i++)                                    //放置数据元素
        L->data[i]=a[i];
    L->length=n;                                        //设置长度
}
```

算法的时间复杂度为 O(n)。

3. 销毁顺序表 L

运算的结果是释放顺序表 L 占用的内存空间。

```
void DestroyList(SqList *&L)
{
 free(L);    //释放线性表占据的所有存储空间
}
```

算法的时间复杂度为 O(1)。

说明:本节采用顺序表指针的目的是通过 malloc 函数分配顺序表的空间,方便使用 free 函数释放其空间。

4. 清空顺序表 L

该运算将顺序表的元素清空,并将顺序表的长度设置为 0。

```
void ClearList(SqList *&L)
{
 L->length=0;    //将顺序表的长度置为 0
}
```

算法的时间复杂度为 O(1)。

5. 求顺序表 L 的长度

该运算返回顺序表 L 的长度,实际只返回 length 域的值即可。

```
int GetLength(SqList *L)
{
return (L->length);
}
```

算法的时间复杂度为 O(1)。

6. 判断顺序表 L 是否为空

该运算返回一个值,表示 L 是否为空表。若 L 为空表,则返回 1,否则返回 0。

```
int IsEmpty(SqList * L)
{
if (L->length==0)
   return 1;
else
   return 0;
}
```

算法的时间复杂度为 O(1)。

7. 输出顺序表

该运算顺序显示 L 中各元素的值。

```
void DispList(SqList * L)
{   int i;
    for(i=0;i<L->length;i++)
      printf(L->data[i]);
}
```

算法中的基本运算采用的是 for 循环的 i++语句,故时间复杂度为 O(n)。

8. 获取顺序表 L 中的某个元素的内容

该运算用 e 返回 L 中第 i 个元素的值。

```
int GetElem(SqList * L,int i,elemtype &e)
{   if (i<1||i>L->length)        //判断 i 值是否合理(1≤i≤L->length),若不合理,则返回 0
       return 0;
    e=L->data[i-1];              //数组中第 i-1 个单元存储着线性表中第 i 个数据元素的内容
    return 1;
}
```

算法的时间复杂度为 O(1)。

9. 在顺序表 L 中检索值为 e 的元素位置

该运算顺序查找第一个值域与 e 相等的元素的逻辑序号。若这样的元素不存在,则返回的值为 0。

```
int LocateElem(SqList * L,elemtype e)
{   for (int i=0;i< L->length;i++)
{
    if (L->data[i]==e)
        return i+1;             //若找到元素 e,则返回其逻辑序号
    else
        return 0;               //若未找到,则返回 0
```

```
    }
}
```

算法中的基本运算采用的是 for 循环的 i++语句,故时间复杂度为 O(n)。


视频讲解

10. 插入数据元素

设顺序表 L=(a_1, a_2, …, a_i, …, a_n),要在其第 i 个位置上插入一个值为 x 的元素,使顺序表变为 L=(a_1, a_2, …, x, a_i, …, a_n)。

算法中要注意以下几点。

- 在向顺序表 L 插入数据元素前,先检查表空间是否已满,在表满的情况下不能再做插入操作,否则将产生溢出错误。
- 检查插入位置是否有效,即 i 的取值是否能被满足,其中有效的取值为 1≤i≤L→length+1,尤其 i 取 L→length+1 时,表示在顺序表最后一个元素的后面插入新元素,这样做是被允许的。
- 待插入的元素 x 若与顺序表 L 中的元素类型不统一,则不允许被插入,但算法中目前无法考察这点,在具体的程序中需要注意到。
- 注意元素逻辑关系上的改变映射到物理位置上的改变。首先为元素 x 的插入腾出地方,于是元素 a_i~a_n这 n−i+1 个元素分别后移一个位置,且 a_n先进行移动;然后 x 进行插入操作;同时修改顺序表的长度,操作完毕。

插入元素时移动元素的过程如图 2.4 所示。

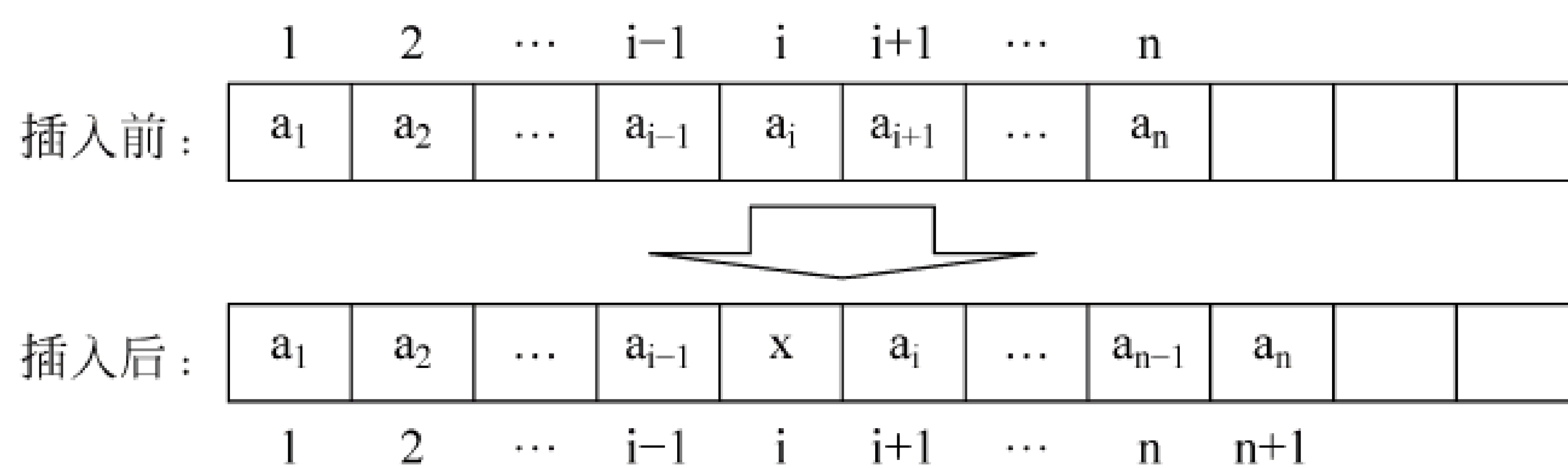


图 2.4 插入元素时移动元素的过程

```
bool ListInsert(SqList *&L,int i,elemtype e)
{
int j;
if(L->length==MaxSize)
    return false;                  //表满错误,返回 false
if(i<1 || i>L->length+1)
    return false;                  //参数错误,返回 false
i--;                               //将顺序表逻辑序号转化为物理序号
for(j=L->length;j>i;j--)           //将 data[i]及后面的元素后移一个位置
    L->data[j]=L->data[j-1];
L->data[i]=e;                      //插入元素 e
L->length++;                       //顺序表长度加 1
return true;                       //成功插入,返回 true
}
```

假设有一个顺序表 L={10,14,20,23,26,35,41},在顺序表的第 4 和第 5 个元素之间插入一个值为 25 的数据元素,则需将第 5~7 个数据元素依次往后移动一个位置。插入元素后顺序表的变化如图 2.5 所示。

数据元素	10	14	20	23	26	35	41
序号	1	2	3	4	5	6	7

插入25(插入于第 4 和第 5 个元素之间)

数据元素	10	14	20	23	25	26	35	41
序号	1	2	3	4	5	6	7	8

图 2.5　插入元素后顺序表的变化

对于该算法来说,元素移动的次数不仅与表长 n=L→length 有关,而且与插入位置 i 有关;当 i=n+1 时,移动次数为 0;当 i=1 时,移动次数为 n,达到最大值。在顺序表 L 中共有 n+1 个可以插入元素的地方。最好情况和最坏情况下时间复杂度分别是 O(1)和 O(n),差别很大,如何衡量算法的时间效率呢?假设 p_i是在第 i 个位置上插入一元素的概率,则在长度为 n 的顺序表中插入一个元素时所需移动元素的平均次数(移动次数的数学期望值)为

$$E_{in}=\sum_{i=1}^{n+1}p_i(n-i+1)$$

默认取等概率的情况,即 $p_i=1/(n+1)$,则

$$E_{in}=\sum_{i=1}^{n+1}p_i(n-i+1)=\sum_{i=1}^{n+1}\frac{1}{n+1}(n-i+1)=\frac{n}{2}$$

所以,顺序表的插入操作约需要移动表中一半的数据元素。设线性表的长度为 n,则算法的时间复杂度为 O(n)。



视频讲解

11. 删除数据元素

设顺序表 $L=(a_1, a_2, \cdots, \mathbf{a_i}, \cdots, a_n)$,要删除其第 i 个位置上的元素,使顺序表变为$L=(a_1, a_2, \cdots, a_{i-1}, a_{i+1}, \cdots, a_n)$。

算法中要注意以下几点。

- 删除顺序表 L 中的数据元素前先检查表空间是否已空,在表空的情况下无法删除操作。
- 检查删除位置是否有效,即 i 的取值是否能被满足,其中有效的取值为 1≤i≤L→length,尤其 i 取 L→length 时,表示删除顺序表中的最后一个元素,这样做就不会产生其他元素的移动。
- 注意元素逻辑关系上的改变映射到物理位置上的改变。首先将待删除的元素放入临时空间中,然后其后续元素依次向前移动“填补”上被删除元素的空缺;同时修改顺序表的长度,操作完毕。

删除顺序表 L 中的第 i 个元素。如果 i 不正确,则显示相应的错误信息;将 a_{i+1}~a_n顺序前移,从而将待删除元素 a_i“挤掉”,因为顺序表要求逻辑上相邻的元素在物理上也相邻。如图 2.6 所示,这样就覆盖了原来的第 i 个元素,达到删除该元素的效果。最后顺序表的长度减 1。

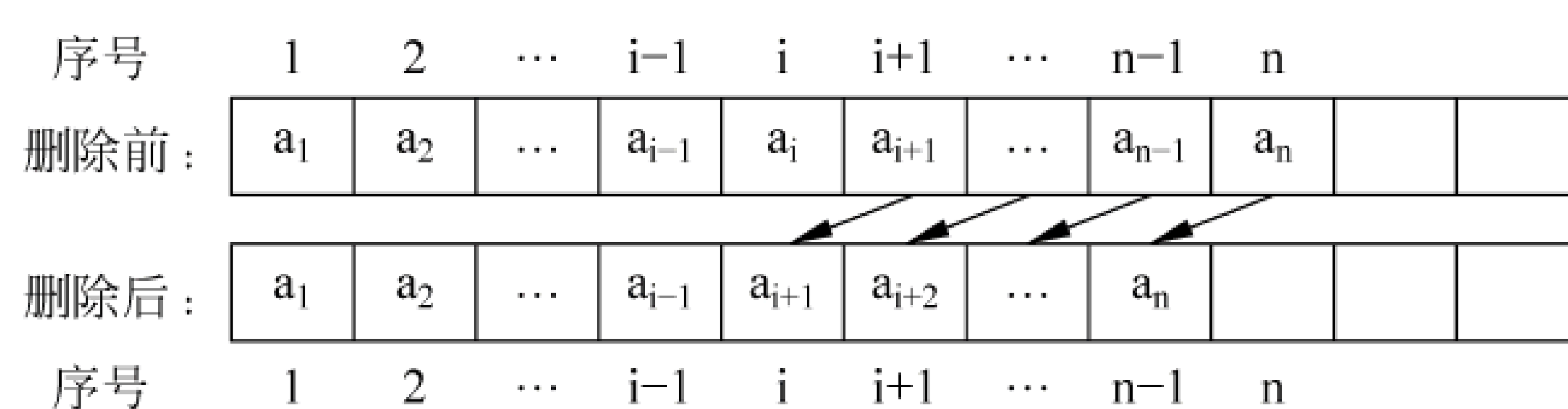


图 2.6　删除元素时移动元素的过程

```
bool ListDelete(SqList *&L,int i, elemtype &e)
{
if(L->length==0)
     return false;                  //表空错误,返回 false
if (i<1 || i>L->length)
     return false;                  //参数错误,返回 false
i--;                                //将顺序表逻辑序号转化为物理序号
e=L->data[i];
for(int j=i;j<L->length-1;j++)      //将 data[i]之后的元素前移一个位置
     L->data[j]=L->data[j+1];
L->length--;                        //顺序表长度减 1
return true;                        //成功删除,返回 true
}
```

假设有一个顺序表 L={10,14,20,23,25,26,35,41}在线性表中删除第 5 个元素 25,则需将第 6~8 个数据元素依次往前移动一个位置。删除数据元素后线性表的变化如图 2.7 所示。

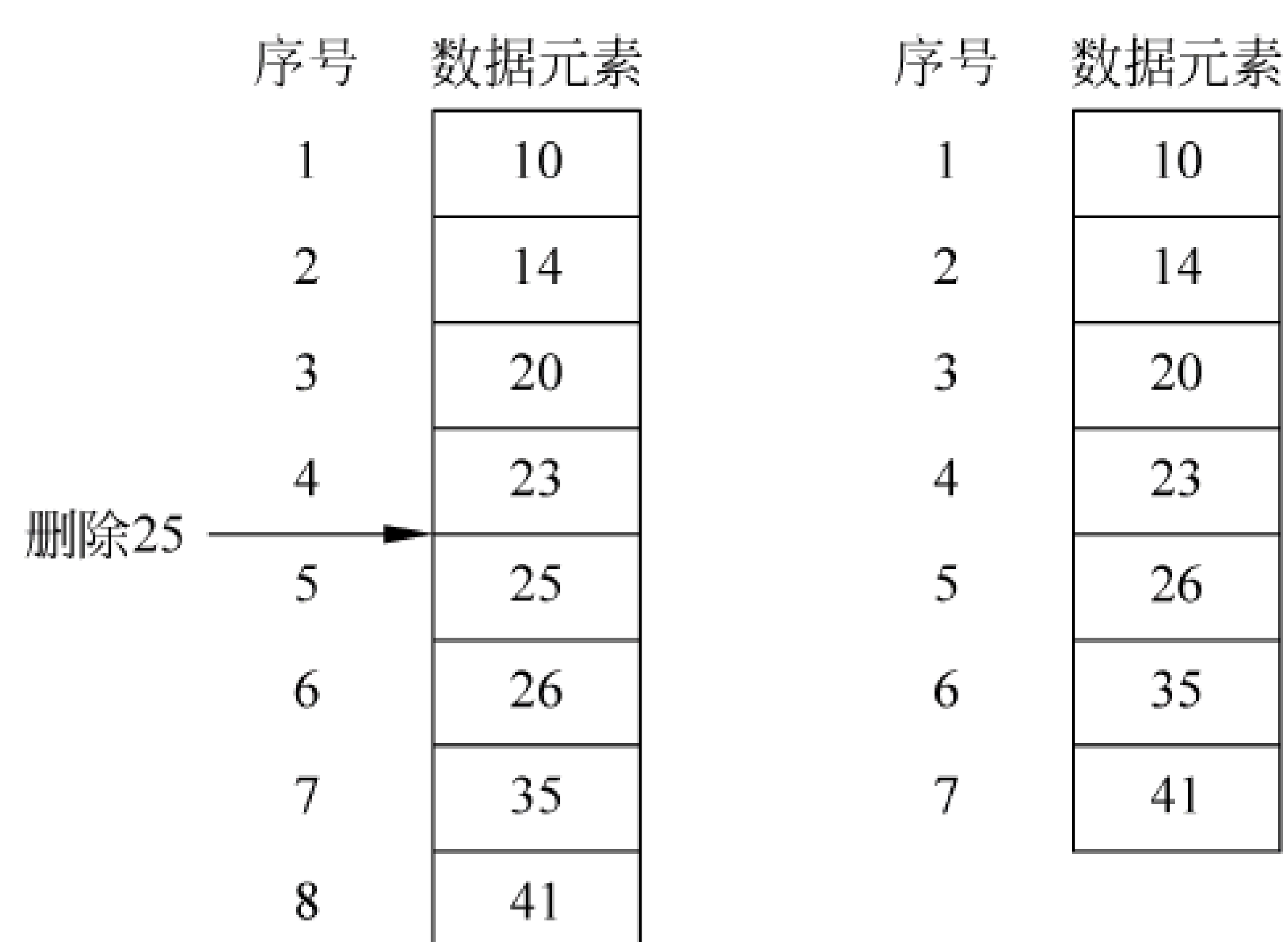


图 2.7　删除数据元素后线性表的变化

如图 2.7 所示,为了删除第 5 个数据元素,必须将第 6~8 个元素都依次往前移动一个位置。

同插入算法相似,元素移动的次数不仅与表长 n=L->length 有关,而且与删除位置 i 有关;当 i=n 时,移动次数为 0;当 i=1 时,移动次数为 n-1,达到最大值。顺序表 L 中共有 n 个可以被删除的元素。最好情况和最坏情况下时间复杂度分别是 O(1)和 O(n),差别很大,如何衡量算法的时间效率呢?假设 q_i是删除第 i 个位置上元素的概率,则在长度为 n 的顺序表中删除一个元素时所需移动元素的平均次数(移动次数的数学期望

值)为

$$E_{dl} = \sum_{i=1}^{n} q_i(n-i)$$

默认取等概率的情况,即 $q_i = 1/n$,则

$$E_{dl} = \sum_{i=1}^{n} q_i(n-i) = \frac{1}{n}\sum_{i=1}^{n}(n-i) = \frac{n-1}{2}$$

所以,顺序表的删除操作约需要移动表中一半的数据元素。设线性表的长度为 n,则算法的时间复杂度为 O(n)。

【例 2.5】 已知一个顺序表 L,其中的元素递增有序排列,设计一个算法,插入一个元素 x(x 为 int 型)后,该顺序表仍然保持递增有序排列(假设插入操作总能成功且元素各不相同)。

分析:由题干可知,解决本题需完成如下两个操作。

step1:找出可以让该顺序表保持有序的插入位置。

step2:将 step1 中找出的位置上以及其后的元素往后移动一个位置,然后将 x 放至腾出的位置上。图 2.8 为元素 12 的插入过程。

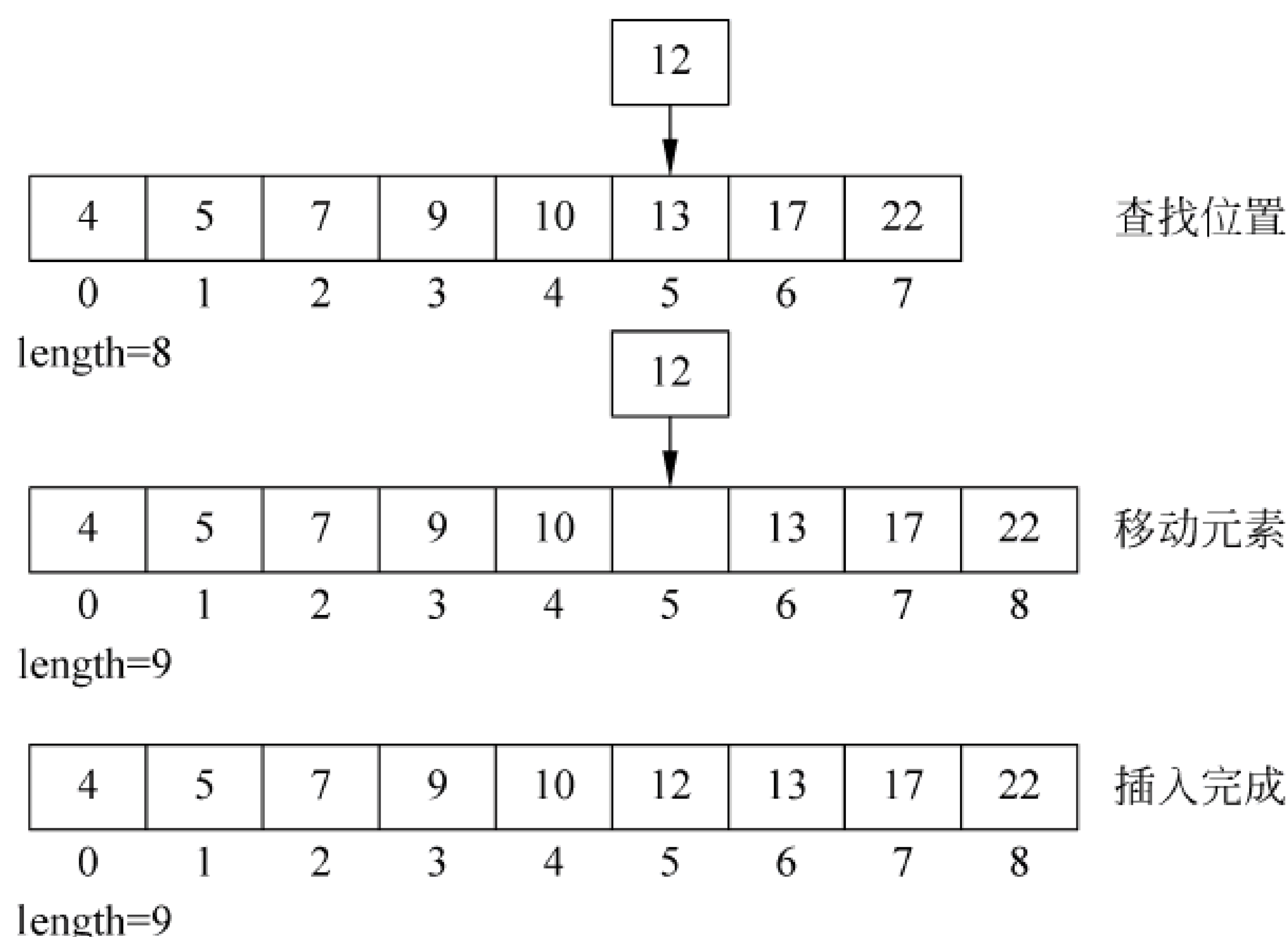


图 2.8　元素 12 的插入过程

操作一:因为顺序表 L 中的元素是递增排列的,所以可以从小到大逐个扫描表中的元素,当找到第一个比 x 大的元素时,将 x 插在这个元素之前即可。如图 2.8 所示,12 为要插入的元素,从左往右逐个进行比较,当扫描到 13 的时候,发现 13 是第一个比 12 大的数,因此 12 应该插在 13 之前。

操作二:找到插入位置后,将插入位置及其以后的元素向后移动一个元素的位置即可。这里有两种移动方法:一种是先移动最右边的元素;另一种是先移动最左边的元素。哪种是正确的移动方法呢?答案是先移动最右边的元素。如果先移动最左边的元素,则右边的元素会被左边的元素覆盖。

操作一的代码如下。

```
int find_Elem(SqList * L, int x)
{
  int i;
  for(i=0;i<L->length;++i)
  {
    if(x<L->data[i])          //对顺序表中的元素从小到大逐个判断
      return i;
   }
return i;
}
```

操作二的代码如下。

```
void insertElem(SqList * &L,int x)
{
 int p,i;
 p=find_Elem(L,x);
 for(i=L->length-1;i>=p;--i)
  {
   L->data[i+1]=L->data[i];         //从右往左,逐个将元素右移一个位置
  }
 L->data[p]=x;                      //将 x 放在插入位置 p 上
 ++(L->length);
}
```

算法的时间主要花费在查找插入位置和移动元素上,时间复杂度为 O(n)。

【例 2.6】 将非递减有序的顺序表 La、Lb 合并成一个新的顺序表 Lc,并保持这种有序性。

分析:由题干可知,解决本题须完成 3 个操作。

step1:初始化空顺序表 Lc。

step2:定义指针 i,j 分别指向 La,Lb 中开始的元素,读取元素进行相应的比较,比较过程分为以下 3 种。

- 若 La 中的元素小于 Lb 中的元素,则将 La 中的元素插入 Lc,同时 i++;
- 若 La 中的元素等于 Lb 中的元素,则将 La 或 Lb 中的元素插入 Lc,同时 i++; j++;
- 若 La 中的元素大于 Lb 中的元素,则将 Lb 中的元素插入 Lc,同时 j++。

step3:讨论顺序表 La 或者 Lb 中元素的剩余情况,若顺序表 La 有剩余,依次将表中剩余的元素插入 Lc 中;若顺序表 Lb 有剩余,则依次将表中剩余的元素插入 Lc 中。

算法代码实现如下。

```
void union(SqList * La, SqList * Lb, SqList * &Lc)
{
int i=0,j=0,k=0;
Lc=(SqList * )malloc(sizeof(SqList));
```

```
Lc→length=0;              //初始化顺序表 Lc
for (;i<La→length&&j<Lb→length;)
{
    if(La→data[i]<Lb→data[j])
      {Lc→data[k]=La→data[i]; i++; k++;}
    else if(La→data[i]==Lb→data[j])
      { Lc→data[k]=La→data[i]; i++; j++;k++;}
    else
      { Lc→data[k]=Lb→data[j]; j++; k++;}
  }
while(i>=La→length)
  { Lc→data[k]=La→data[i]; i++; k++;}
while(i>=La→length)
  { Lc→data[k]=Lb→data[j]; j++; k++;}
Lc→length=La→length+Lb→length;
}
```

算法的时间复杂度为 O(ListLength(La)+ListLength(Lb))。

2.3 线性表的链式存储结构

对长度变化较大的线性表,预先分配空间须按最大空间分配时,会有空间利用不充分、线性表的容量扩充困难的问题。克服缺点的办法是采用链式存储结构。本节将讨论链式存储结构及其基本运算的实现。



视频讲解

2.3.1 线性表的链式存储结构——链表

线性表的链式存储结构是指用一组任意的存储单元(可以连续,也可以不连续)存储线性表中的数据元素。为了反映数据元素之间的逻辑关系,对于每个数据元素,不仅要表示它的具体内容,还要附加一个表示它的直接后继元素存储位置的信息。假设有一个线性表(春、夏、秋、冬),可用表 2.2 所示的形式存储。

表 2.2 链式存储示意表

存储地址	内容	直接后继存储地址
100	夏	120
…	…	…
120	秋	160
…	…	…
144	春	100
…	…	…
160	冬	NULL
…	…	…

为了表示每个数据元素 $a_i(1\leqslant i\leqslant n)$ 与其直接后继数据元素 a_{i+1} 之间的逻辑关系,对数据元素来说,除了存储其本身的信息之外,还需存储一个指示其直接后继的信息(即直接后

继的存储位置)。这两部分信息组成数据元素 a_i 的存储映像,称为**结点(node)**。它包括两个域:其中存储数据元素 a_i 信息的域称为**数据域**;存储直接后继 a_{i+1} 存储位置的域称为**指针域**。指针域中存储的信息称为**指针或链**。结点的示意图如图 2.9 所示。

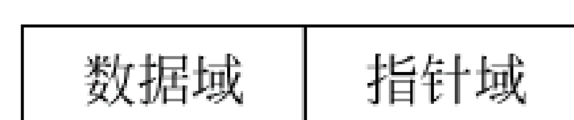


图 2.9　结点的示意图

链表的每个结点中只包含一个指针域,这样构成的链接表称为线性单向链接表,简称**单链表**。根据指针域的个数,链表又可以分为单链表、双链表和多链表。在每个结点中,除数值域外,设置两个指针域,分别指向前驱结点和后继结点,这样构成的链接表称为线性双向链接表,简称**双链表**。

例如,一年有 4 个季节,在计算机中采用链式存储映像如表 2.2 所示。通常,将链表简化成箭头相链接的结点序列,结点之间的箭头表示链域中的指针。设定头指针 Head 指示第一个元素"春"所在的地址,于是链表被简化成如图 2.10(a)所示的形式,因为在使用链表时,我们只关心其所表示的线性表中数据元素之间的逻辑顺序,而不是每个数据元素在存储器中的实际位置。

在线性表的链式存储中,为了便于插入和删除算法的实现,每个链表附加一个头结点,并通过头结点的指针唯一表示该链表。从该指针所指的头结点出发,沿着结点的链(即指针域的值)可以访问到每一个结点。图 2.10(b)是带头结点的链表结构示意图。

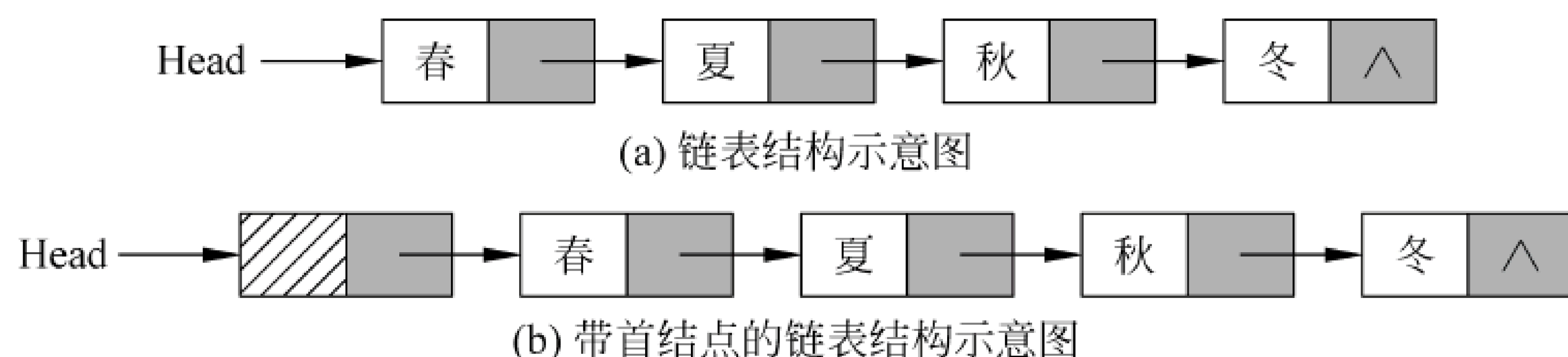


图 2.10　表 2.2 对应的链表结构

线性表($a_1,a_2,\cdots,a_i,\cdots,a_n$)采用链式存储方式形成链表。对链表的任何操作,都必须从第一个结点开始,沿着每个结点向后的指针域可以访问到每个结点。单链表 Head 和双链表 DHead 如图 2.11 所示。

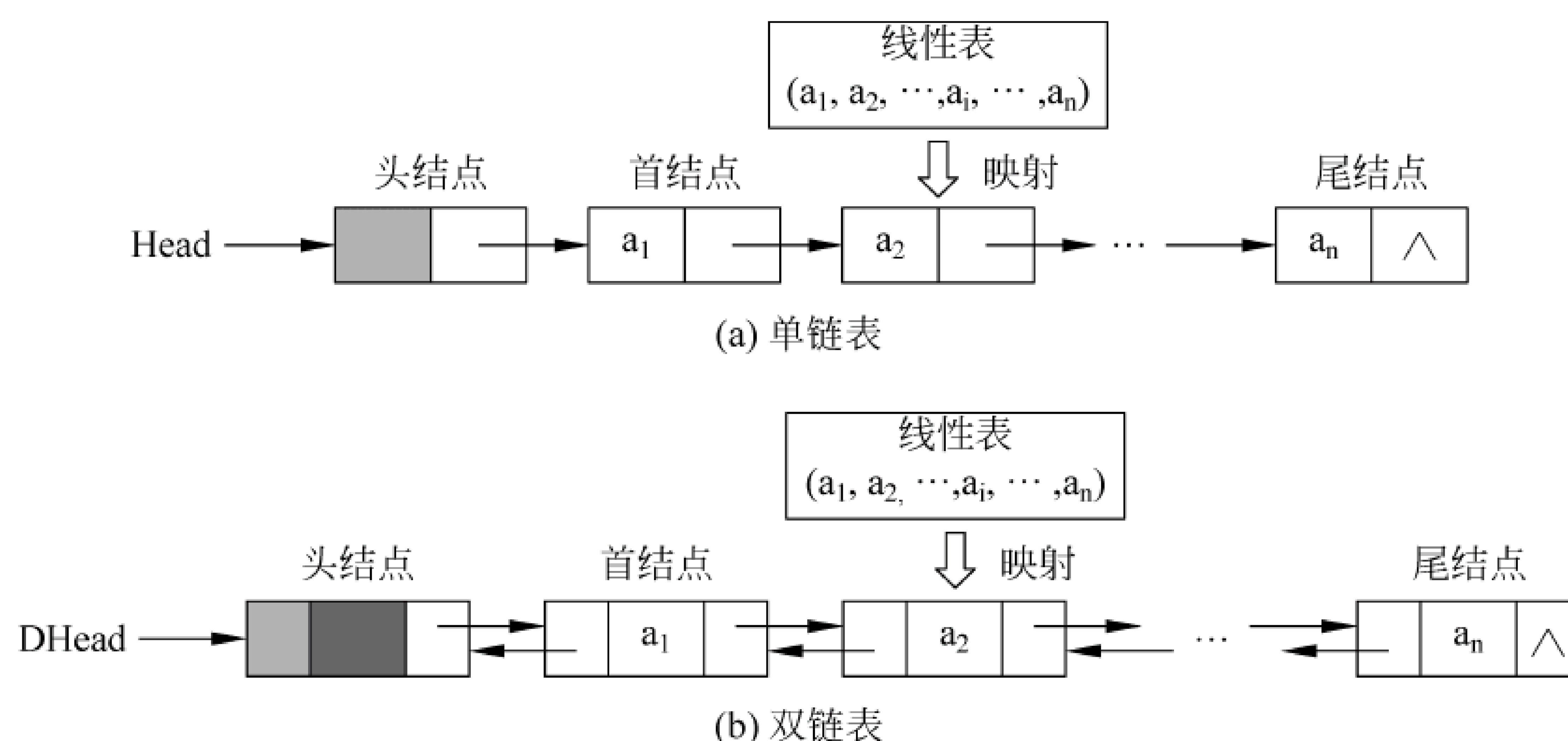


图 2.11　链表示意图

注意:

- 在链表中设置头结点有什么好处?

答：头结点即在链表的首结点之前附设的一个结点，该结点的数据域中不存储线性表的数据元素，其作用是在对链表进行操作时，可以对空表、非空表的情况以及对首结点进行统一处理，使编程更方便。**默认的单链表是带头结点的**。

- 如何表示空表？

答：无头结点时，当头指针的值为空时表示空表，即 Head==NULL；有头结点时，当头结点的指针域为空时表示空表，即 Head→next==NULL。无论带表头与否，空链表的长度都为 0。

在单链表中，假定每个结点的类型用 LNode 表示，它应包含存储元素的数据域，这里用 data 表示，其类型用通用类型标识符 elemtype 表示，还包括存储后继结点位置的指针域，这里用 next 表示。链表类型用 LinkList 表示，类型的定义如下。

```
typedef struct LNode
{   elemtype data;          //数据域
    struct LNode * next;    //指针域
}LNode, * LinkList;         //结点和链表的类型名
```

应用示例：

```
LNode a;            //变量 a 用于存储单链表中的一个结点
LNode * p ;         //p 为指向结点(结构)的指针变量(简称 p 结点)
LinkList Head;      //单链表 Head(头指针即为 Head)
```

链式存储结构的特点如下。

- 线性表中的数据元素在存储单元中的存放顺序与逻辑顺序不一定一致。
- 在对线性表操作时，只能通过头指针进入链表，并通过每个结点的指针域向后扫描其余结点，这样就会造成寻找第一个结点和寻找最后一个结点所花费的时间不等，具有这种特点的存取方式被称为**顺序存取方式**。

链表运算中，常用的库函数如下。

```
sizeof(类型名)        //获取类型所占内存字节数
malloc(m)            //申请一段 m 字节长度的地址空间，并返回这段空间的首地址
free(p)              //释放指针 p 所指的存储空间，即将存储空间归还给系统
```

注意：凡是动态申请的内存，使用完毕后必须释放，否则将产生内存泄漏。

在线性表的顺序存储结构中，由于逻辑上相邻的两个元素在物理位置上紧邻，则每个元素的存储位置都可从线性表的起始位置计算得到。而在单链表中，任何两个元素的存储位置之间都没有固定的联系，然而，每个元素的存储位置都包含在其直接前驱结点的信息中。假设 p 是指向线性表中第 i 个数据元素(结点 a_i)的指针，则 p→next 是指向第 $i+1$ 个数据元素(结点 a_{i+1})的指针。换句话说，若 p→data=a_i，则 p→next→ data=a_{i+1}。由此，在单链表中，取得第 i 个数据元素必须从头指针出发寻找，故单链表是表，是非随机存取的存储结构。

2.3.2 单链表基本运算的实现

单链表中,每个结点都有一个指针域指向其后继结点。在进行结点插入和删除时,不能简单地只对该结点进行操作,还需要考虑其前后的结点。

同样,为了操作简单,假设 elemtype 为 int 类型,使用如下自定义类型语句。

```
typedef int elemtype;
```

1. 线性链表的初始化

建立一个空的单链表,分为创建带头结点的单链表和不带头结点的单链表。带头结点的单链表需要申请头结点空间,将指针域设置为 NULL,返回头结点的地址;不带头结点的单链表直接返回空指针作为头指针,如图 2.12 所示。



图 2.12 初始化链表 Head

1) 带头结点链表的初始化

```
LinkList InitList_L( )
{ LinkList Head;
  Head=(LNode *)malloc(sizeof(LNode));
  Head→next = NULL;
  return (Head);
}
```

2) 不带头结点链表的初始化

```
LinkList InitList_L( )
{
  return NULL;
}
```

默认的链表是带头结点的。

2. 插入和删除结点操作



视频讲解

在单链表中插入和删除结点是最常用的操作,它是建立单链表和实现相关基本运算的基础。

1) 插入结点操作

插入运算是将值为 x 的新结点插入单链表的第 i 个位置上。假设将元素 x 插入值为 a_{i-1} 和 a_i 的结点之间,实现的步骤如下。

step1:搜索 a_i 的前驱结点,定义指针 p=头指针,计数器 j=0;当(p 非空)且(j<i-1)时,p 后移,++j;

step2:p 定位到第 i-1 个结点对应的位置,元素 x 生成新结点 s。

step3:插入新结点 s。

在单链表 Head 中插入结点如图 2.13 所示。

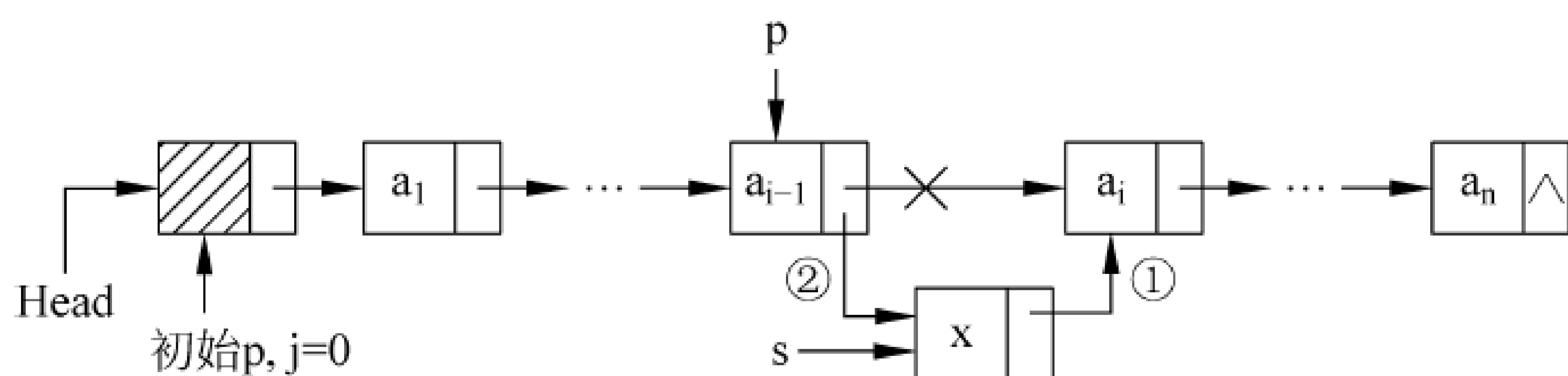


图 2.13　在单链表 Head 中插入结点

插入算法的具体实现过程如下。

```
bool ListInsert_L(LinkList &L, int i, elemtype X)
{
LNode *p=L;                //p 的初始值指向头结点
int j=0;
while(p && j<i-1)          //搜索 i 位的前驱(即 i-1 位结点)
  { p=p->next;
    ++j;
  }
if(!p||j>i-1)
    return false;
s=(LinkList)malloc(sizeof(LNode));
s->data=X;                 //生成新结点
s->next=p->next;
p->next=s;                 //新结点 s 插到 p 结点的后面
return true;
}
```

注意：在 p 结点的后面插入 s 结点时，不需要元素发生位置的移动，只需要修改 2 个指针，且指针修改的次序不能随意改变，即 s→next＝p→next；p→next＝s；次序不能更改为 p→next＝s；s→next＝p→next；否则无法实现插入操作。

2）删除结点操作

删除运算是将单链表的第 i 个结点删去。从头结点“数”到 a_i 的前驱 p 结点，将 p 结点的 next 域连到 a_i 的后继结点。实现步骤如下。

step1：“数”到要删除结点的前驱结点，定义指针 p＝头指针，计数器 j＝0，若 p 的后继非空且未到 a_i 的前驱，则重复 p 后移，++j；($1\leqslant i\leqslant n$，j 与下标同值)。

step2：搜索后，若(p 的后继＝＝空)或(j＜i－1)，则返回删除位置错误提示。

step3：删除结点，将 p 结点的指针域指向 p 的后继的后继。释放结点(p→next)所用空间。

在单链表 Head 中删除结点如图 2.14 所示。

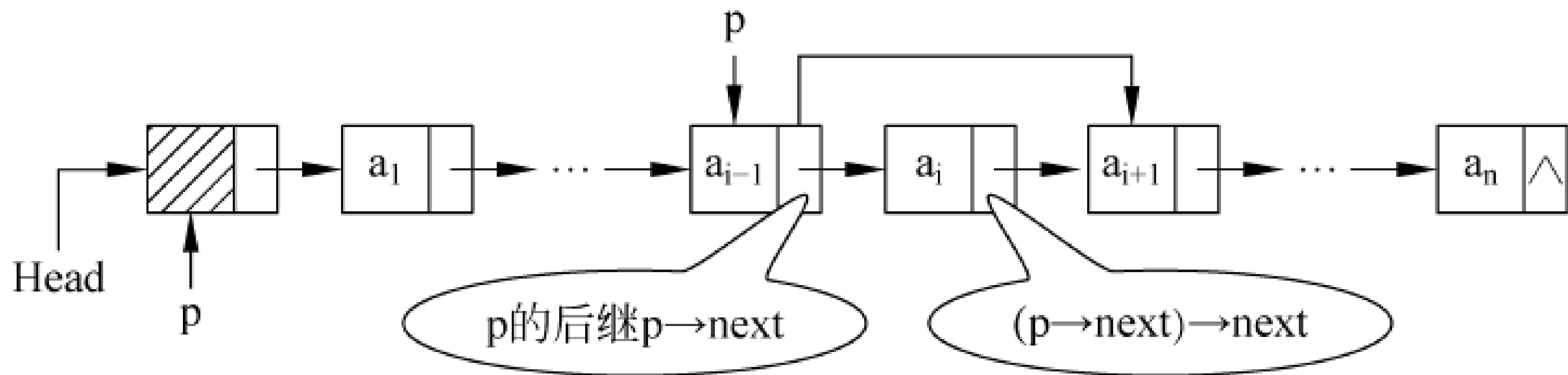


图 2.14　在单链表 Head 中删除结点

删除算法的具体实现过程如下。

```
bool ListDelete_L(LinkList &L, int i)
{ LNode *q, *p=L; int j=0;              //指针 p 指向头结点，计数器 j=0
  while (p->next && j<i-1)
    { p->next;
      ++j;
    }
    if(!(p->next)||j>i-1)
      return false;
  q=p->next;                            //指针 q 指向要删除的结点
  p->next=q->next;
  free(q);
  return true;
}
```

注意：删除 p 结点的后继结点 q 时，不需要元素发生位置的移动，只需要修改 1 个指针即可。

从上面的算法可知，链表的插入和删除算法时间复杂度都为 O(n)，n 是链表的长度。

3. 创建链表

在进行单链表的基本运算之前，必须先建立单链表。建立单链表的常用方法有如下两种。



视频讲解

1）首插法建表

该方法从一个空链表开始，读取字符数组 a[ ]中的字符，生成新结点，将读取的数据存放到新结点的数据域中，然后将新结点插入到当前链表的表头上(即头结点的后面)，直到读完字符数组 a[ ]的所有元素为止。采用首插法建表的算法如下。

```
void CreateListF(LinkList &L, elemtype a[ ], int n)
{
    LNode *s;
    int i;
    L=(LinkedList)malloc(sizeof(LinkList));
    L->next=NULL;             //初始化空链表 L
    for(i=0;i<n;i++)          //循环建立数据结点
    {
     s=(LNode *)malloc(sizeof(LNode));
     s->data=a[i];            //生成新结点 s
     s->next=L->next;         //新结点 s 插到头结点的后面
     L->next=s;
    }
}
```

算法的时间复杂度为 O(n)，其中 n 为单链表中数据结点的个数。若数组 a[ ]包含 4 个数组元素 1,2,3 和 4，则调用 CreateListF(L,a,4)建立单链表的过程如图 2.15 所示。

2）尾插法建表

首插法建立链表虽然算法简单，但生成的链表中结点的次序和原数组元素的次序相反。

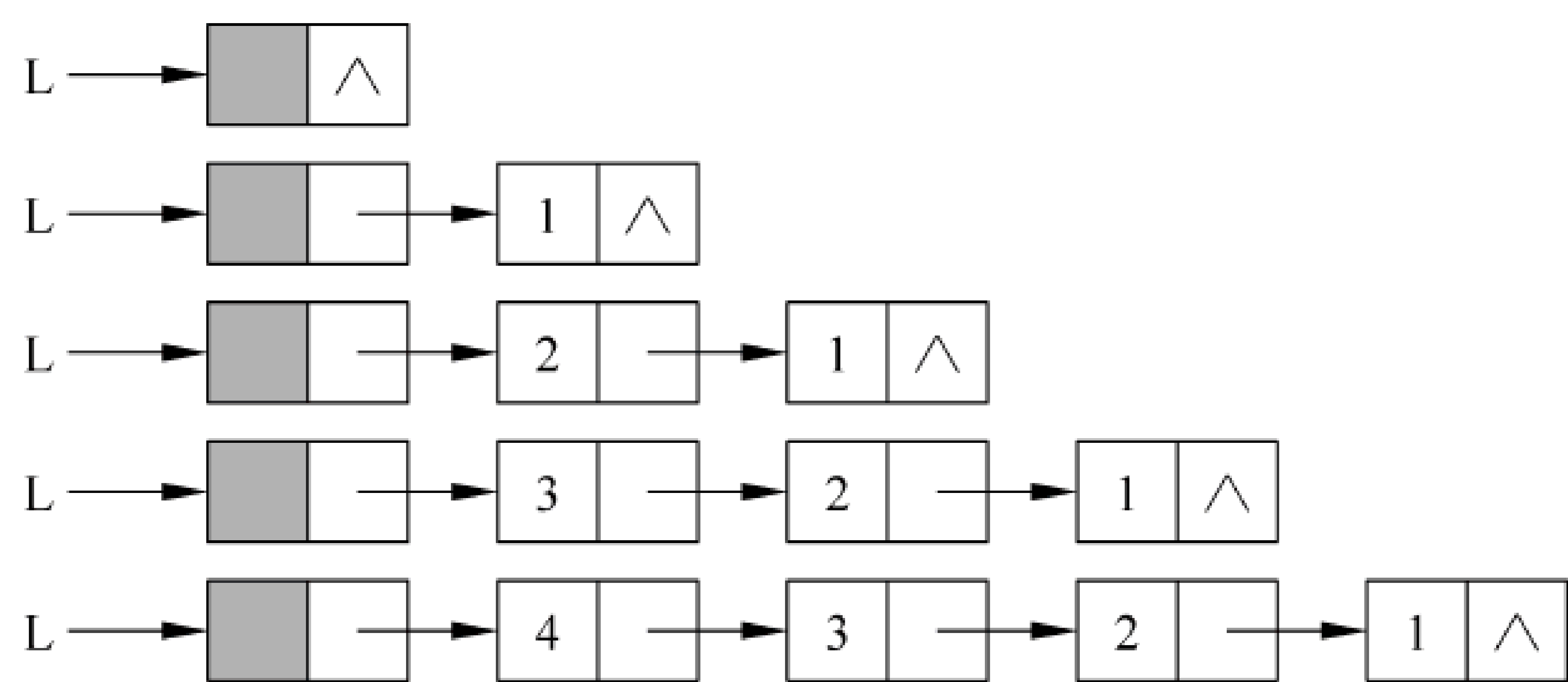


图 2.15　头插法建立单链表 L

若希望两者次序一致，可采用尾插法建立。该方法是将新结点插到当前链表的表尾，为此必须增加一个尾指针 r，使其始终指向当前链表的尾结点。采用尾插法建表的算法如下。

```
void CreateListR(LinkList &L, elemtype a[ ], int n)
{
    LNode  *s, *r;
    int i;
    L=(LinkList)malloc(sizeof(LinkList));          //创建头结点
    r=L;                                           //r 始终指向尾结点，开始时指向头结点
    for(i=0;i<n;i++)                               //循环建立数据结点
{
    s=(LNode *)malloc(sizeof(LNode));
    s->data=a[i];                                  //创建新结点 s
    r->next=s;                                     //将 s 结点插到 r 结点后面
    r=s;                                           //跟踪新的尾结点
}
  r->next=NULL;                                    //将尾结点的 next 域置为 NULL
}
```

算法的时间复杂度为 O(n)，其中 n 为单链表中数据结点的个数。若数组 a 包含 4 个元素 1,2,3 和 4，则调用 CreateListR(L,a,4)建立的单链表如图 2.16 所示。

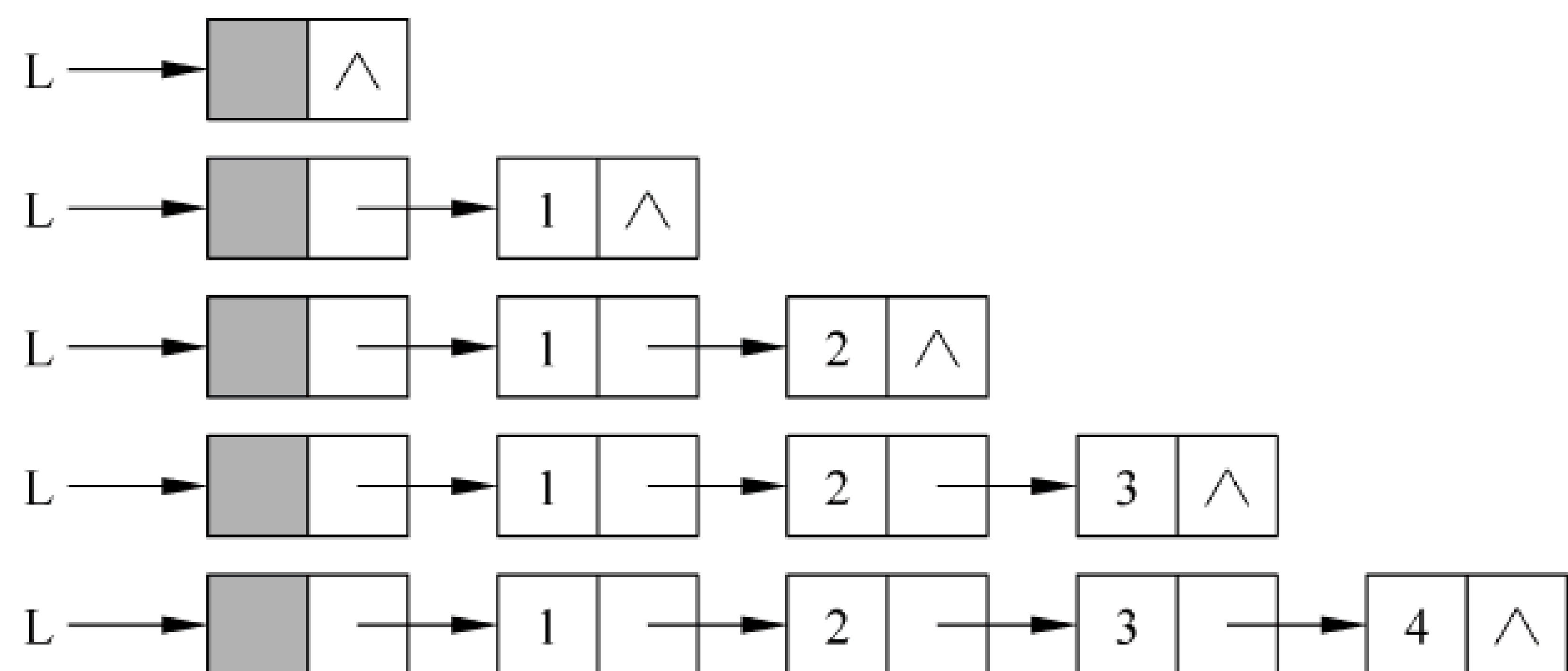


图 2.16　尾插法建立单链表 L

4. 计算链表长度

计算单链表 L 中的数据结点的个数：从第一个结点开始，一个结点一个结点计数，直至链表尾。

```
int ListLength(LinkList L)
{
  LNode *p=L;            //p 指向头结点
  int n=0;               //头结点的序号为 0
  while(p->next!=NULL)
{ n++;
  p=p->next;
}
  return n;
}
```

算法的时间复杂度为 O(n),其中 n 为单链表中数据结点的个数。对于不带头结点的链表,空表的情况要单独处理。

5. 在链表 L 中检索值为 e 的数据元素

在单链表 L 中从头开始找第一个值域与 e 相等的结点,若存在这样的结点,则返回其逻辑序号,否则返回 0。

```
int LocateElem(LinkList L, elemtype e)
{
  LNode *p=L->next;                    //p 指向开始结点
  int i=1;                             //将开始结点的序号置为 1
  while(p!=NULL && p->data==e)         //查找 data 值为 e 的结点,其序号为 i
  {
    p=p->next;
    i++;
  }
  if (p==NULL)                         //若不存在元素值为 e 的结点,则返回 0
    return 0;
  else
    return i;                          //若存在元素值为 e 的结点,则返回其逻辑序号 i
}
```

本算法的时间复杂度为 O(n),其中 n 为单链表中数据结点的个数。

6. 清空链表

释放单链表 L 占用的内存空间,即逐一释放全部结点的空间。

```
void DestoryList(LinkList L)
{ LNode *pre=L, *p=L->next;            //pre 指向 p 结点的前驱结点
  while (p!=NULL)                      //扫描单链表 L
{ free(pre);                           //释放 pre 结点
  pre=p;                               //pre 与 p 同步后移一个位置
  p=pre->next;
  }
  free(pre);                           //释放尾结点
}
```

本算法的时间复杂度为 O(n)，其中 n 为单链表中数据结点的个数。



【例 2.7】 已知带头结点的单链表 L(非空链表)，设计算法实现链表逆置。要求不能额外增加存储空间。

分析：链表逆置后，之前的第一个结点变成最后一个结点，第二个结点变成倒数第二个结点，以此类推；根据单链表中指针指示的单方向性，链表的逆置不适合于数据元素之间的交换，即将之前链表第一个结点的元素值和最后一个结点的元素值交换，第二个结点的元素值和倒数第二个结点的元素值交换等；我们发现逆置操作将原链表重新创造了一次，于是从创建链表的方法思考，首插法刚好满足了该问题的需求。操作步骤如下。

step1：定义指针 p 指示链表中的结点，同时将链表 L 置为空表。

step2：使用首插法不断地将 p 结点插到新链表 L 中，从而实现了链表的逆置。

算法的实现过程如下。

```
void reverse(LinkList &L)
{
 LNode *p=L->next, *q=p->next;          //p 指示链表中的结点
 L->next=NULL;                          //将链表置为空
 while(p)
 {
 p->next=L->next;
 L->next=p;
 p=q;
 q=q->next;
}
}
```

算法的时间复杂度为 O(n)，其中 n 为单链表中数据结点的个数。

【例 2.8】 已知带头结点的单链表 L(非空链表)，设计算法实现查找链表的倒数第 k 个结点。若查找成功，则返回指示该结点的指针，否则返回 NULL。

分析：从单链表的尾结点倒序查找是无法实现的，因为单链表中无向前指示的指针。如何高效率地实现查找定位呢？可设定两个指针 p，q 分别定位到链表的首结点，首先让 p 指针先移动 k−1 次，然后 q 指针再和 p 指针同步依次后移，当 p 指针到达链表尾结点时，此刻的 q 指针到达倒数第 k 个结点，返回 q 指针即可。

算法的实现过程如下。

```
void GetLink(LinkList L, int k)
{
 LNode *p=L->next, *q=L->next;
 int i=1;
 if(!p)
  return p;          //若链表为空，则返回 NULL
 for(;i<k;i++)
{
 p=p->next;
```

```
    }                       //p指针定位在第k个结点
    while(p→next)
    {
    q=q→next;
    p=p→next;
    }
    return q;               //q返回倒数第k个结点
}
```

算法的时间复杂度为O(n),其中n为单链表中数据结点的个数。此外,还有其他的实现方法。例如,先设定一个指针p,从链表的开头到结尾计算其长度,此时时间消耗为O(n),然后再将问题转化成查找第n－k＋1个结点,p指针再次从链表开头出发进行查找,这样,最坏情况下最多遍历两趟即可查找到要求的结点,时间复杂度仍然为O(n);对比以上两种方法,显然第一种情况稍好一些。

【例2.9】 A和B是两个带头结点的单链表,其中元素递增有序。设计一个算法,将A和B归并成一个元素值非递减有序的链表C。要求不能额外增加存储空间。

分析:提出问题表面上是要求实现链表的合并操作,而实际上可以认为是创建链表C的过程,并且在创建的过程中要使得链表C同链表A和链表B保持性质的一致性;因此考虑到使用尾插法创建链表C。操作步骤如下。

step1:初始化链表C,并定义指针 * pa, * pb分别从链表A和B的头部出发。

step2:比较pa结点与pb结点的元素值。比较结果分3种情况。

- pa→data＜pb→data,将pa结点插到链表C的尾部。
- pa→data＝＝pb→data,将pa结点与pb结点均插到链表C的尾部。
- pa→data＞pb→data,将pb结点插到链表C的尾部。

step3:重复步骤**step2**,链表A或链表B中的所有结点均插到链表C。

step4:将链表A或B的剩余部分接到链表C的尾部。

算法的实现过程如下。

```
void merge(LinkList A, LinkList B, LinkList &C)
{
  LNode *pa=A→next, *pb=B→next, *r;
  C=A;                  //用A的头结点做C的头结点
  C→next=NULL;
  free(B);              //B的头结点已经没用,释放掉
  r=C;                  //r跟踪链表C的尾部
  while(pa!=NULL && pb!=NULL)
  {
   if(pa→data<=pb→data)
   {
    r→next=pa;
    pa=pa→next;
    r=r→next;
```

```
        }
       else
       {
        r->next=pb;
        pb=pb->next;
        r=r->next;
        }
    }
    if(pa)                    //链表 A 有剩余
     r->next=pa;
    if(pb)                    //链表 B 有剩余
     r->next=pb;
    }
```

算法的时间复杂度为 O(n+m)，其中 n 是链表 A 的结点个数，m 是链表 B 的结点个数。试想，若提出将链表 A 与链表 B 合并成非递增有序的链表 C(有序性刚好与链表 A、B 相反)，则可以使用首插法创建链表 C。

【例 2.10】 有一个带头结点的单链表 $L=(a_1,b_1,a_2,b_2,\cdots,a_n,b_n)$，设计算法将其拆分成两个带头结点的单链表 L1 和 L2，$L1=(a_1,a_2,\cdots,a_n)$，$L2=(b_n,b_{n-1},\cdots,b_2,b_1)$。

分析：提出问题表面上是要求实现链表的拆分操作，而实际上可以认为是创建链表 L1 和 L2 的过程，观察 L1 和 L2 中结点次序的特点，将链表 L 中的结点从头开始"采摘"下来，第一个结点尾插到链表 L1 中，第二个结点首插到链表 L2 中，依次交替进行，直至链表 L 中的结点"采摘"完毕。操作步骤如下。

step1：初始化链表 L1 和 L2，L1 可以使用链表 L 的头结点。

step2：定义指针 *p 指示链表 L 的开始结点，指针 *r 指示链表 L1 的尾部。

step3：先将 p 结点插到 L1 表中，再将 p 的后继结点插到 L2 中。

step4：重复步骤 **step3**，直至 p 为空。

算法的实现过程如下。

```
void spList(LinkList L, LinkList &L1, LinkList &L2)
{
 LNode *p=L->next, *r, *q;
 L1=L;
 r=L1;
 L2=(LinkList)malloc(sizeof(LinkList));
 L2->next=NULL;
 while(p)
{
 r->next=p;                 //尾插法将 p 结点插到 L1
 p=p->next;
 q=p->next;
 p->next=L2->next;          //首插法将 p 结点插到 L2
 L2->next=p;
 p=q;
```

```
    }
    r→next=NULL;            //将链表 L1 的尾结点 next 置空
}
```

算法的时间复杂度为 O(n),其中 n 与链表 L 的结点个数有关(约一半)。通过上面的几个例子发现,创建链表的方法应用比较广泛,可用于链表逆置、合并、拆分,甚至排序等方面。

2.3.3 双链表

在单链表中,通过一个结点找到它的后继结点非常方便,操作的时间复杂度为 O(1),但要找到它的前驱结点却很麻烦,操作的时间复杂度为 O(n),因为只能从链表的头指针开始,沿着每个结点的 next 域查找,究其原因是单链表的各个结点只指向其后继结点的指针域 next,如果希望查找前驱的时间复杂度也是 O(1),可以为每个结点多添加一个指针域 prior 指示结点的前驱,使得链表可以双向查找。采用这种结点结构组成的链表称为**双链表**。

双链表中的每个结点都有两个指针域:一个指向其后继结点,另一个指向其前驱结点。双链表的结点结构示意图如图 2.17 所示。



图 2.17　双链表的结点结构示意图

双向链表结点的类型定义如下。

```
typedef struct DNode
{  elemtype data;
   struct DNode  * prior;        //指向前驱结点
   struct DNode  * next;         //指向后继结点
} * DLinkList;
```

和单链表类似,双链表也是由头指针唯一确定的,且默认也是带头结点的。双链表虽然比单链表多占了存储空间,但它给运算带来了便利。例如,双链表进行搜索遍历时,可以从链表的任意位置向前移或向后移。图 2.18 给出了空双链表和非空双链表。

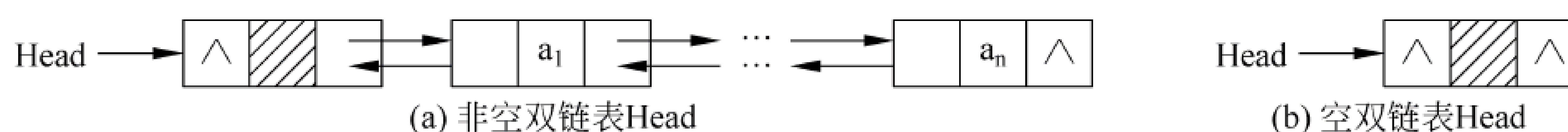


图 2.18　双链表

在双链表中,如访问某个特定结点、计算长度等操作时,仅涉及一个方向的指针,算法描述过程和单链表的操作类似;但在链表的创建、结点的插入和删除操作时有较大区别,原因在于增加了一个指针,所以增加了操作环节。下面着重讨论这些操作。

1. 建立双链表

建立双链表也有两种方法。与首插法建立单链表的过程相似,采用首插法建立双链表的算法如下。

1）首插法建立双链表

```
void CreateList(DLinkList &L, elemtype a[ ], int n)
{
    DNode  * s;
    int i;
    L=(DLinkList)malloc(sizeof(DLinkList));
    L→prior=L→next=NULL;            //初始化双链表
    for(i=0;i<n;i++)                //循环建立数据结点
{
    s=(DNode  * )malloc(sizeof(DNode));
    s>data=a[i];
    s→next=L→next;                  //将新结点 s 插到头结点之后
    if(L→next!=NULL)
     L→next→prior=s;
    L→next=s;
    s→prior=L;
}
}
```

2）尾插法建立双链表

```
void CreateList(DLinkList &L, elemtype a[ ], int n)
{
 DNode  * s, * r;
 int i;
 L=(DLinkList)malloc(sizeof(DLinkList));
 r=L;                    //r 始终指向尾结点，开始时指向头结点
 for(i=0;i<n;i++)        //循环建立数据结点
{
 s=(DNode  * )malloc(sizeof(DNode));
 s→data=a[i];
 r→next=s;
 s→prior=r;              //将新结点 s 插到尾结点 r 之后
 r=s;                    //跟踪新的尾结点
}
r→next=NULL;             //将尾结点 next 域置为 NULL
}
```

2. 插入和删除操作

1）插入结点操作

插入运算是将值 x 的新结点插到双链表 L 的第 i 个位置上。假设将 x 插到值为 a_{i-1} 和 a_i 的结点之间，实现的步骤如下。

step1：搜索 a_i 的前驱结点，定义指针 p=头指针，计数器 j=0；当(p 非空)且(j<i−1)时，p 后移，++j；。

step2：p 定位到第 i−1 个结点对应的位置，元素 x 生成新结点 s。

step3：插入新结点 s 并修改相关指针域。

双链表插入操作如图 2.19 所示。

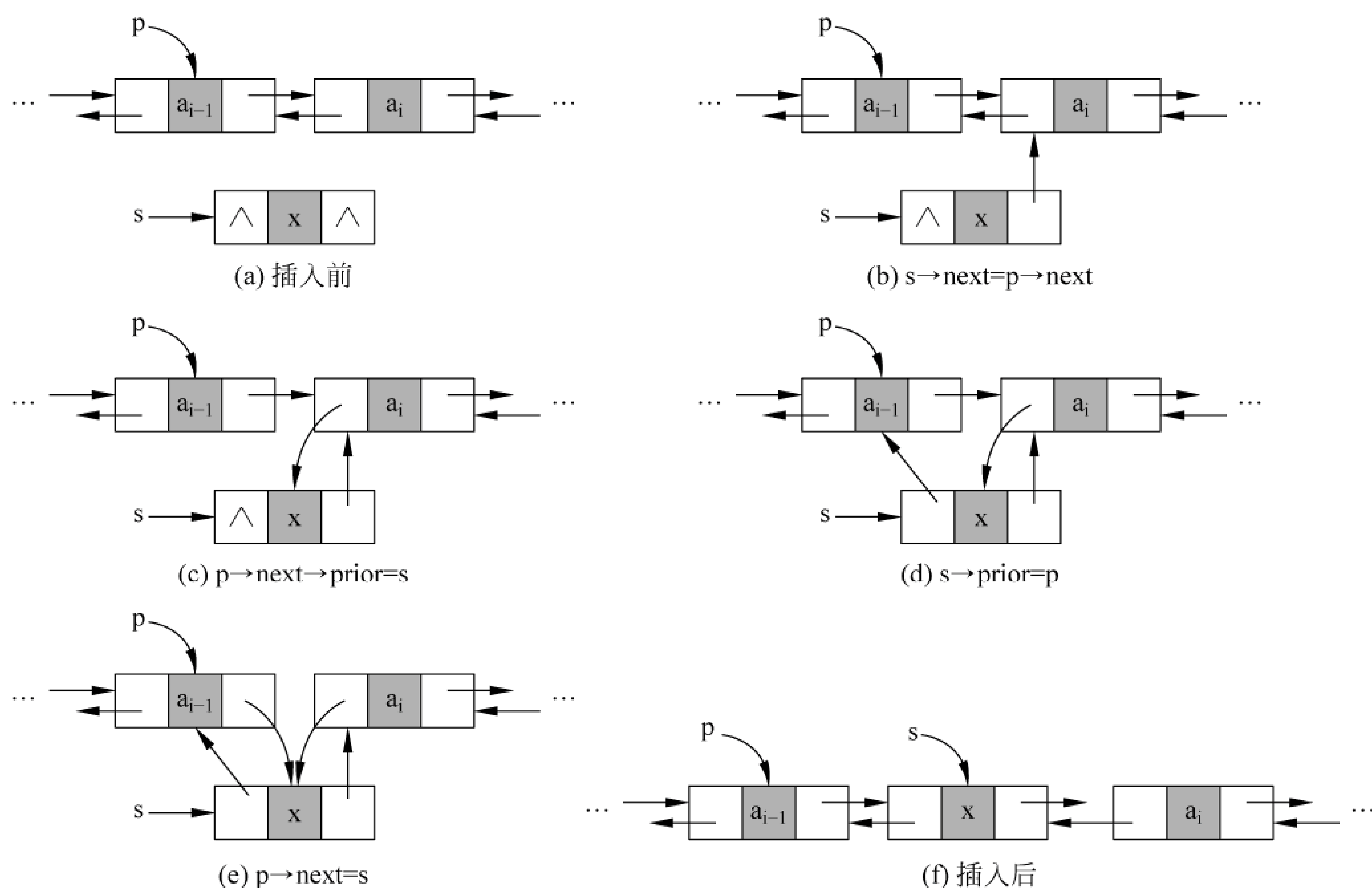


图 2.19　双链表插入操作

算法的实现过程如下。

```
bool ListInsert(DLinkList &L, int i, elemtype e)
{
int j=0;
DLNode * p=L, * s;              //p 指向头结点,j 置为 0
while(j<i-1 && p!=NULL)         //查找第 i-1 个结点
{ j++;
  p=p->next;
}
if(!p)                          //若未找到第 i-1 个结点,则返回 false
  return false;
else                            //若找到第 i-1 个结点 p,则在其后插入新结点 s
{
s=(DLNode * )malloc(sizeof(DLNode));
s->data=e;                      //创建新结点 s
s->next=p->next;                //在 p 结点之后插入 s 结点
if(p->next!=NULL)               //若 p 结点存在后继结点,则修改其前驱指针
     p->next->prior=s;
s->prior=p;
p->next=s;
return true;
}
}
```

本算法的时间复杂度为 O(n)，其中 n 为双链表中数据结点的个数。在双链表中插入一个结点须修改 4 个指针，且语句的次序不能随意更改。

2）删除结点操作

删除运算是将双链表的第 i 个结点删去。从头结点"数"到 a_i的前驱 p 结点，将 p 结点的 next 域连到 a_i的后继结点 a_{i+1}，a_{i+1}结点的 prior 域连到 p 结点。实现的步骤如下。

step1："数"到要删除结点的前驱结点，定义指针 p＝头指针，计数器 j＝0，若 p 的后继非空且未到 a_i的前驱，则重复 p 后移++j；(1≤i≤n，j 与下标同值)。

step2：搜索后，若(p 的后继＝＝空)或(j＜i－1)，则删除位置错误。

step3：删除结点，将 p 结点的指针域指向 p 的后继的后继。释放结点(p→next)所用的空间。

双链表删除操作如图 2.20 所示。

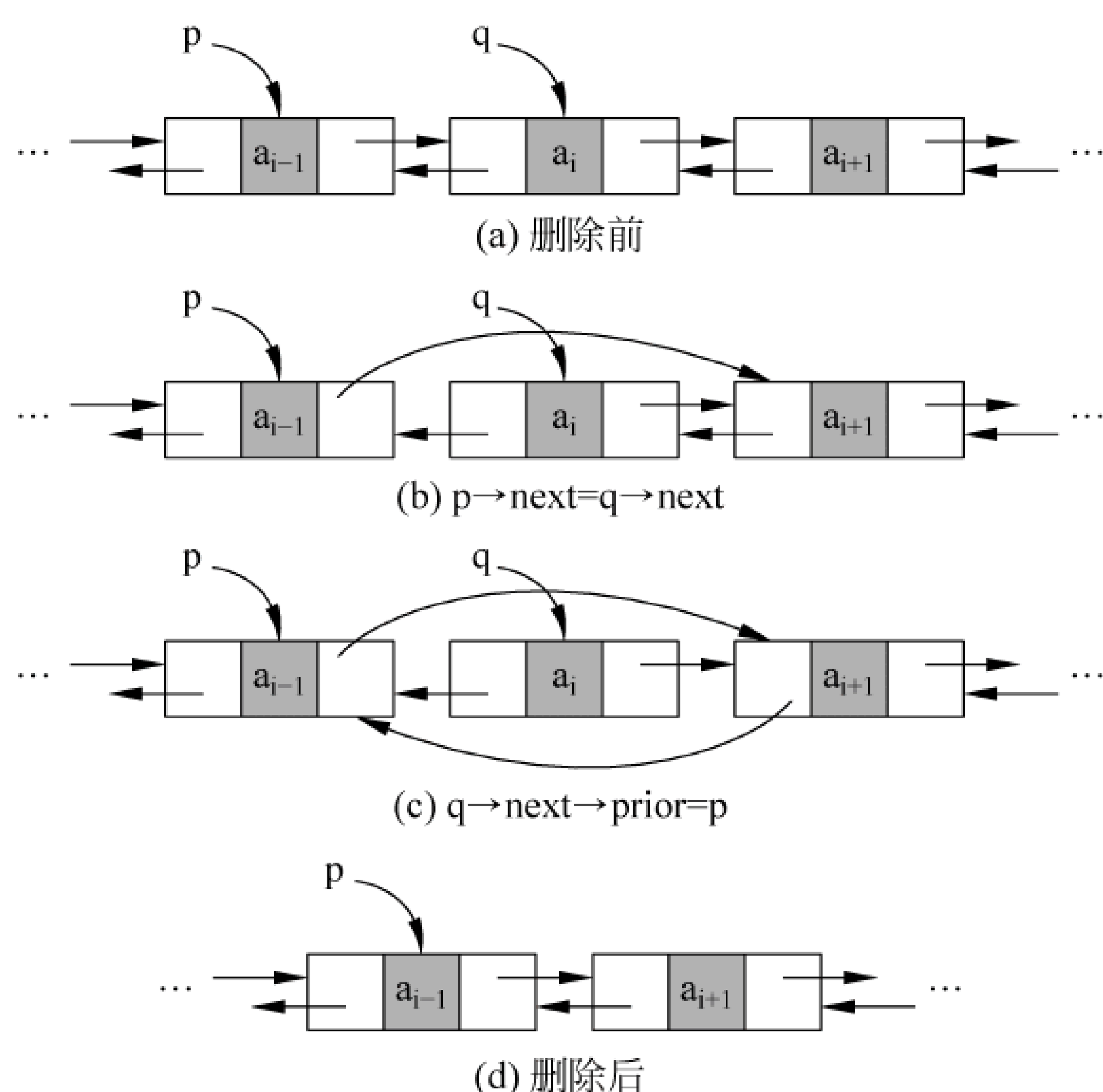


图 2.20　双链表删除操作

算法的实现过程如下。

```
bool ListDelete(DLinkList &L, int i)
{
int j=0;
DLNode *p=L, *q;                  //p 指向头结点,j 置为 0
while(j<i-1 && p!=NULL)           //查找第 i-1 个结点
{ j++;
  p=p->next;
}
if(!p)                            //若未找到第 i-1 个结点,则返回 false
  return false;
else                              //找到第 i-1 个结点 p
```

```
{
q=p->next;                      //q 指向第 i 个结点
if(q==NULL)                     //当不存在第 i 个结点时,返回 false
    return false;
p->next=q->next;                //从双链表中删除 q 结点
if(p->next!=NULL)               //若 p 结点存在后继结点,则修改其前驱指针
    p->next->prior=p;
free(q);
return true;
}
}
```

算法的时间复杂度为 O(n),其中 n 为双链表中数据结点的个数。在双链表中删除一个结点须修改 2 个指针,显然与插入不同,删除语句的执行与次序无关。

2.3.4 循环链表

循环链表(circular Linked list)是另一种形式的链式存储结构。它的特点是表中最后一个结点的指针域指向头结点,整个链表形成一个环。由此,从表中任一结点出发均可找到表中的其他结点,如图 2.21 所示。类似地,可以有多重链的循环链表。

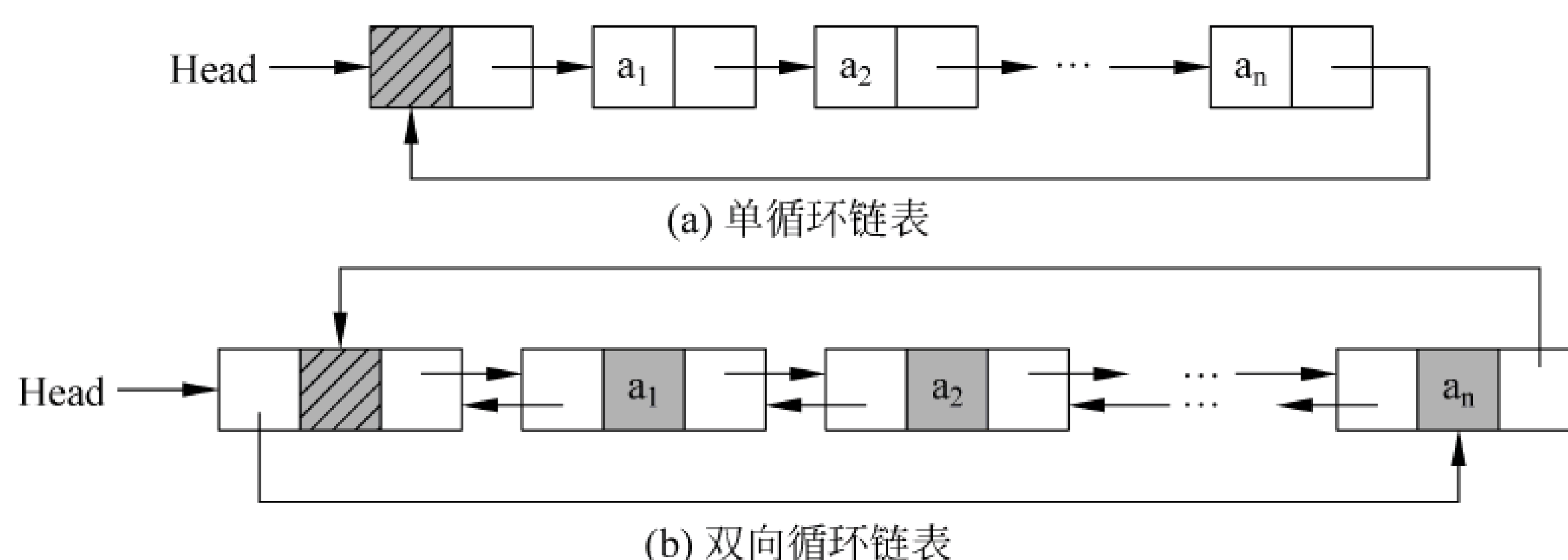


图 2.21 循环链表

循环链表的操作和非循环链表的类型定义及操作实现基本相同,只是在条件的判定上做了改变。循环链表的特点如下。

- 从任一结点出发均可找到表中的其他结点。
- 操作仅有一点与单链表不同。循环条件如下。

  单链表:p==NULL 或 p->next==NULL

  循环链表:p==Head 或 p->next==Head
- 循环链表默认也带表头。

特例:带头结点的空循环链表样式,如图 2.22 所示。

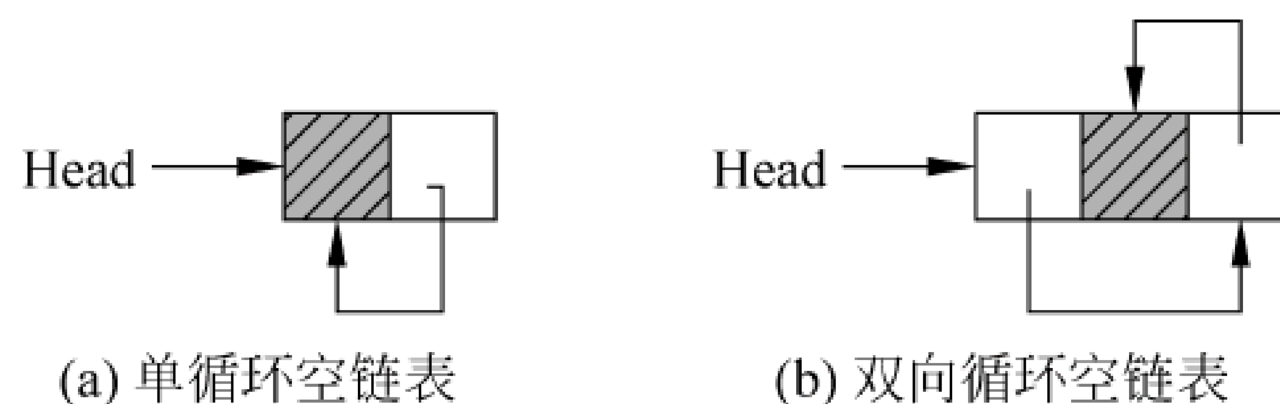


图 2.22 循环空链表

【例 2.11】 一个带头结点的单链表 L,设计算法将其改变为单循环链表 L。

分析:实现过程和计算单链表 L 的长度基本相同,当 p 指向链表的尾结点时,将尾结点的 next 域链接到头结点即可。

算法的实现过程如下。

```
void CircleList(LinkList &L)
{
 LNode *p=L;
 while(p->next)              //将指针 p 定位到尾结点
    p=p->next;
 p->next=L;                  //将尾结点的 next 域连到头结点
}
```

算法的时间复杂度为 O(n),其中 n 为链表中的结点个数。

【例 2.12】 一个带头结点的循环单链表 L,设计算法计算链表的长度。

算法的实现过程如下。

```
void GetLen(LinkList L)
{
 LNode *p=L;
 int len=0;
 while(p->next!=L)         //判断指针 p 是否到达表尾
  {
  p=p->next;
  len++;
  }
  return len;
}
```

算法的时间复杂度为 O(n),其中 n 为链表中的结点个数。

【例 2.13】 一个带头结点的循环双链表 L,设计算法删除第一个 data 值等于 x 的结点。

分析:实现过程类似之前讲过的链表的查找操作,定义指针 p 从链表头部出发,沿着结点的 next 域向后查找,依次将找到的结点值和 x 对比,若相等,则再做删除操作并返回 true;若从头到尾没有发现值与 x 相等的结点,则返回 false。

算法的实现过程如下。

```
bool del(DLinkList &L, elemtype e)
{
 DLNode *p=L->next;
 while(p!=L&&p->data!=x)
    p=p->next;
if(p!=L)                 //找到第一个值为 x 的结点
 {
  p->next->prior=p->prior;
  p->prior->next=p->next;
```

```
        free(p);
        return true;
    }
    else
        return false;
}
```

2.3.5 STL与链表

C++标准模板库中提供了一个链表类 list,它是众多容器的一种。这个链表类提供了功能强大的函数,用户利用这些函数可以方便地对链表进行操作。在使用链表类 list 之前,要在文件开始处包含如下代码。

```
#include <list>
```

相对而言,vector 是连续性空间,而 list 就较为复杂。链表类 list 支持对结点进行插入与删除操作,每次插入或删除一个元素,就配置或释放一个元素空间。这些操作既可以在表尾进行,也可以在表头进行。STL 中的 list 就是一个双向链表,可高效率地插入和删除元素。下面是 list 结点的结构定义。

```
template
struct _list_node{
typedef void * void_pointer;
void_pointer next;
void _pointer prev;
T data;
}
```

list 本身和 list 的结点是不同的结构,需要分开设计。例如,定义一个 STL 的 list。

```
#include <iostream>
#include <string>
#include <list>
using namespace std;
int main( )
{
    list<int> mylist1;
    mylist1.push_front(5);
    cout<<mylist1.size( )<<endl;              //链表长度为 1
    list<string>mylist2(10);
    cout<<mylist2.size( )<<endl;              //链表长度为 10
    list<double>mylist3(2,3.14);              //初始值为 3.14
    cout<<mylist3.back( )<<endl;
    mylist3.pop_back( );
```

```
cout<<mylist3.empty( )<<endl;
 return 0;
}
```

上面代码中,函数 size()与 empty()分别用来计算链表长度与判断链表是否为空。为了能方便地使用 STL 链表类 list,STL 提供了丰富的成员函数。下面通过简单的示例说明如何使用主要成员函数。

```
assign()                                给 list 赋值
back()                                  返回最后一个元素
begin()                                 返回指向第一个元素的迭代器
clear()                                 删除所有元素
empty()                                 如果 list 是空的,则返回 true
end()                                   返回末尾的迭代器
erase()                                 删除一个元素
front()                                 返回第一个元素
get_allocator()                         返回 list 的配置器
insert(iterator,10)
插入一个元素 10 到元素迭代其 iterator 之前,一般 iterator=find(list.begin(),list.end(),3)
max_size()                              返回 list 能容纳的最大元素数量
merge(list<T> &x)                       将 x 合并到 *this
pop_back()                              删除最后一个元素
pop_front()                             删除第一个元素
push_back()                             在 list 的末尾添加一个元素
push_front()                            在 list 的头部添加一个元素
begin()                                 返回指向第一个元素的逆向迭代器
remove()                                从 list 删除元素
remove_if()                             按指定条件删除元素
rend()                                  指向 list 末尾的逆向迭代器
resize()                                改变 list 的大小
reverse()                               把 list 的元素倒转
size()                                  返回 list 中的元素个数
sort()                                  给 list 排序
splice(iterator position,list &x)       //list.splice(position,list2)
                                        //将 list2 合并到 list 中的 position 之前
splice(iterator position,list &x,iterator i)  //将元素插到 list 中的 position 之前
splice(iterator position,list &x,iterator first,iterator last)
                                        //将 first — last 之间的元素插到 list 中的 position 之前
swap()                                  //交换两个 list
unique()                                //删除 list 中重复的元素
```

假设两个 list 对象 c1,c2 分别为 c1(10,20,30),c2(40,50,60)。还有一个迭代器 it = list::iterator 用来指向 c1 或 c2 元素。

1. list 的构造函数和析构函数

```
list<Elem>c;                            //构造函数,产生一个空 list,没有任何元素
```

```
list<Elem>c(c2);                     //copy 构造函数,建立 c2 的同类型 list,并成为 c2 的一份副本
list<Elem>c = c2;                    //copy 构造函数,建立一个新的 list 作为 c2 的副本
list<Elem>c(rv);                     //move 构造函数,建立一个新的 list,取 rvalue rv 的内容
                                     //(始自 C++11)
list<Elem>c = c2;                    //move 构造函数,建立一个新的 list,取 rvalue rv 的内容
                                     //(始自 C++11)
list<Elem>c(initlist);               //建立一个 list,以初值列 initlist 的元素为初值(始自 C++11)
list<Elem>c = initlist;              //建立一个 list,以初值列 initlist 的元素为初值(始自 C++11)
list<Elem>c(n);                      //利用元素的 default 构造函数生成一个大小为 n 的 list
list<Elem>c(n,elem);                 //建立一个含 n 个元素的 list,值都是 elem
list<Elem>c(c1.begin(),c1.end());    //c 含 c1 一个区域的元素[First, _Last)
c.~list();                           //销毁所有元素,释放内存
```

2. list 的非易型操作

```
c.empty()        //返回容器是否为空(相当于 size()==0,但也许较快)
c.size()         //返回目前的元素个数
c.max_size()     //返回元素个数的最大可能量
c1 == c2         //返回 c1 是否等于 c2(对每个元素调用==)
c1 != c2         //返回 c1 是否不等于 c2(相当于!(c1 == c2))
c1 > c2          //返回 c1 是否大于 c2
c1 >= c2         //返回 c1 是否大于等于 c2
c1 < c2          //返回 c1 是否小于 c2
c1 <= c2         //返回 c1 是否小于等于 c2
```

3. assign()给 list 赋值

```
c.assign(n,elem);        //复制 n 个 elem,赋值给 c
c.assign(beg,end);       //将区间[beg,end)内的元素赋值给 c
c.assign(initlist);      //将初值列 initlist 的所有元素赋值给 c
```

4. swap()交换两个 list

```
c1.swap(c2);       //置换 c1 和 c2 的数据
swap(c1,c2);       //置换 c1 和 c2 的数据
```

5. list 元素直接访问

```
front() 返回第一个元素 (不检查是否存在第一个元素)
int i=c1.front();        //i=10
back() 返回最后一个元素 (不检查是否存在最后一个元素)
int i=c1.back();         //i=30
```

6. 迭代器相关函数

```
begin() 返回一个迭代器,指向第一个元素
```

```
it=c1.begin();                                          //*it=10

end() 返回一个迭代器,指向最后一个元素的下一位置
it=c1.end();                                            //*(--it)=30;

rbegin() 返回一个反向迭代器，指向反向迭代器的第一个元素
list<int>::reverse_iterator riter=c1.rbegin();          //*riter=30

rend() 返回一个反向迭代器，指向反向迭代器最后一个元素的下一个位置
list<int>::reverse_iterator riter=c1.rend();            //*(--riter)=10

cbegin() 返回一个 const 迭代器，指向第一个元素
list<int>::const_iterator citer=c1.cbegin();            //*citer=10 且为 const

cend() 返回一个 const 迭代器，指向最后一个元素的下一个位置
list<int>::const_iterator citer=c1.cend();              //*(--citer)=30 且为 const

crbegin() 返回一个 const 反向迭代器，指向反向迭代器第一个元素
list<int>::const_reverse_iterator criter=c1.crbegin();  //*criter=30 且为 const

crend() 返回一个 const 反向迭代器，指向反向迭代器最后一个元素的下一个位置
list<int>::const_reverse_iterator criter=c1.crend();    //*(--criter)=10 且为 const
```

7. clear()删除所有元素

```
c1.clear();              //c1 为空,c1.size 为 0
```

8. erase()：删除一个元素

```
c1.erase(pos);          //移除 pos 位置上的元素,返回下一个元素的位置
c1.erase(beg,end);      //移除区间[beg,end)内的所有元素,返回下一个元素的位置
```

9. remove()：从 list 中删除元素

```
c.remove(val);          //移除所有其值为 val 的元素
```

10. remove_if()：按指定条件删除元素

```
c1.remove_if(op);       //移除所有"造成 op(elem)结果为 true"的元素
```

11. resize()：改变 list 的大小

```
c1.resize(num)          //将元素数量改为 num(如果 size()变大,多出来的新元素都需以 default
                        //构造函数完成初始化)
c1.resize(num,elem)     //将元素数量改为 num(如果 size()变大,多出来的新元素都是 elem 的副本)
```

12．insert()：插入一个元素到 list 中

```
c.insert(pos,elem);          //在 iterator 位置 pos 之前插入一个 elem 副本,并返回新元素的位置
c1.insert(pos,n,elem);       //在 iterator 位置 pos 之前插入 n 个 elem 副本,并返回第一个新元素
                             //的位置(或返回 pos--,如果没有新元素)
c1.insert(pos,beg,end);      //在 iterator 位置 pos 之前插入区间[beg,end)内所有元素的一份副本,
                             //并返回第一个新元素的位置(或返回 pos--,如果没有新元素)
c1.insert(pos,initlist);     //在 iterator 位置 pos 之前插入初值列 initlist 内所有元素的一份副本,
                             //并返回第一个新元素的位置(或返回 pos--,如果没有新元素)
```

例如：

```
c1.insert(++c1.begin(),100);                        //c1(10,100,20,30)
c1.insert(c1.begin(),2,200);                        //c1(200,200,20,30);
c1.insert(++c1.begin(),c2.begin(),--c2.end());//c1(10,40,50,20,30);
c1.insert(++c1.begin(),{100,200});                  //c1(10,100,200,20,30)
```

13．emplace()：插入以 args 为初值的元素

```
c.emplace(pos,args...)
//在 iterator 位置 pos 之前插入一个以 args 为初值的元素,并返回新元素位置
c.emplace_back(args...)
//在末尾插入一个以 args 为初值的元素,不返回任何东西(始自 C++11)
c.emplace_front(args...)
//在起点插入一个以 args 为初值的元素,不返回任何东西(始自 C++11)
```

14．pop_back()：删除最后一个元素，但是不返回

```
c1.pop_back();       //c1(10,20);
```

15．pop_front()：删除第一个元素，但是不返回

```
c1.pop_front();      //c1(20,30)
```

16．push_back()：在 list 的末尾添加一个元素

```
c1.push_back(100);       //c1(10,20,30,100)
```

17．push_front()：在 list 的头部添加一个元素

```
c1.push_front(100);       //c1(100,10,20,30)
```

18．merge()：合并两个 list 并使之默认升序（也可改）

```
c.merge(c2)
//假设 c 和 c2 容器都包含 op()准则下的已排序元素,则将 c2 中的全部元素转移到 c,并保证合并后
//的 list 仍已排序
```

```
c.merge(c2,op)
//假设 c 和 c2 容器都包含已排序元素,则将 c2 中的全部元素转移到 c,并保证合并后的 list 在 op()
//准则下已排序
```

例如:

```
c2.merge(c1);                     //c1 现为空,c2 现为 c2(10,20,30,40,50,60)
c2.merge(c1,greater<int>());      //同上,但 c2 现为降序
```

19. reverse():把 list 的元素倒转

```
c1.reverse();        //c1(30,20,10)
```

20. sort():给 list 排序,默认升序(可自定义)

```
c.sort()        //以 operator<为准则对所有元素排序
c.sort(op)      //以 op()为准则对所有元素排序
```

例如:

```
c1.sort();                      //c1(10,20,30)
c2.sort(greater<int>());        //同上,但 c1 现为降序
```

21. splice():合并两个 list

```
c.splice(pos,c2)               //将 c2 内的所有元素转移到 c 之内,迭代器 pos 之前
c.splice(pos,c2,c2pos)         //将 c2 内的 c2pos 所指的元素转移到 c 内 pos 之前
c.splice(pos,c2,c2beg,c2end)
//将 c2 内的[c2beg,c2end)区间内的所有元素转移到 c 内 pos 之前
```

例如:

```
c1.splice(++c1.begin(),c2);                    //c1(10,40,50,60,20,30);c2 为空   全合并
c1.splice(++c1.begin(),c2,++c2.begin());       //c1(10,50,20,30); c2(40,60)   指定元素合并
c1.splice(++c1.begin(),c2,++c2.begin(),c2.end());
//c1(10,50,60,20,30); c2(40)   指定范围合并
```

22. unique():删除 list 中重复的元素

```
c.unique()        //如果存在若干相邻而数值相同的元素,就移除重复元素,只留一个
c.unique(op)      //如果存在若干相邻元素都使 op()的结果为 true,则移除重复元素,只留一个
```

例如:

```
c1.unique();        //假设 c1 开始为(-10,10,10,20,20,-10),则之后为 c1(-10,10,20,-10)
```

2.4 综合案例

2.4.1 一元多项式的表示及相加运算

1. 问题描述

两个多项式 AH,BH 如下所示。

$$AH = 1 - 10 \cdot x^{6} + 2 \cdot x^{8} + 7 \cdot x^{14}$$

$$BH = -x^{4} + 10 \cdot x^{6} - 3 \cdot x^{10} + 8 \cdot x^{14} + 4 \cdot x^{18}$$

一般情况下,一元多项式可写成:

$$Pn(x) = p1x^{e1} + p2x^{e2} + \cdots + pmx^{em}$$

2. 解题思路

通过观察,不难发现每一个多项式都是由若干个单项式相加构成的,单项式的通项 pix^{ei}是系数为 pi、指数为 ei 的项。其中,pi 的值非零且 $0 \leqslant e1 \leqslant e2 \leqslant \cdots \leqslant em$;因此,多项式的表示即为若干单项式的表示,在通常应用中,多项式的指数可能很高并且变化幅度很大,因此,使用顺序存储结构很难确定其存储规模大小,在此考虑使用链式存储结构表示多项式,即一个多项式被存储为一个链表,简单的表达是采用单链表定义,这样组成多项式的单项式被表示为一个一个的结点,根据每个单项式的特点可使用一个一个的二元组表示。例如,$2x^{8}$可定义为(2,8)二元组表示,$7x^{14}$可定义为(7,14)表示,即每个单项式 pix^{ei}可定义为二元组(pi,ei),因此设定单项式对应的结点类型如图 2.23 所示。

coef	exp	next

图 2.23 一元多项式链表的结点结构图

单项式结点的定义如下。

```
typedef struct {
    float coef;              //系数 pi
    int exp;                 //指数 ei
    struct ployn * next;
} ployn;
```

综上所述,多项式 AH,BH 对应的链表如图 2.24 所示。

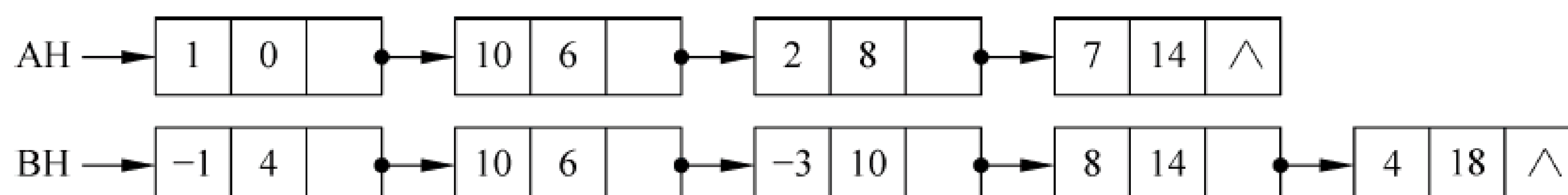


图 2.24 多项式 AH,BH 对应的链表

两个多项式相加转化为链表 AH,BH 的合并运算,设结果为单链表 CH,如图 2.25 所示。

- 指数不同的项,按指数值大小,从小到大依次链入链表 CH 中。
- 指数相同的项,将系数相加,若相加结果是零,则该项相加后消失,两链表指针同时后移一位;否则创建新结点,其系数值为两单项式系数相加的结果,指数不变,两链表指针同时后移一位。

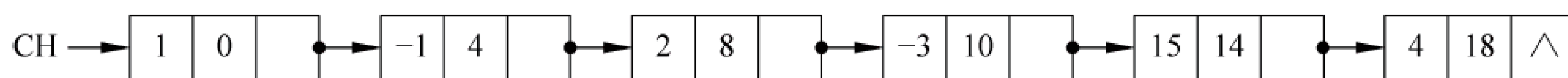


图 2.25　结果多项式对应的链表

3. 代码实现

```
void addployn(ployn * ah, ployn * bh, ployn * &ch) {
  //创建并初始化链表 ch
  ch = (ployn * )malloc(sizeof(ployn));
  ch->next = NULL;
  ployn * pa = ah, * pb = bh, * pc = ch, * s, * qa, * qb;
  while (pa != NULL && pb != NULL) {
    //若指数不同,则按照指数的大小依次将单项式链入链表
    if (pa->exp < pb->exp) {
      //尾插法插入链表 ch,pc 的作用是尾指针
      pc->next = pa;
      pc = pa;
      pa = pa->next;
    } elseif (pa->exp > pb->exp) {
      pc->next = pb;
      pc = pb;
      pb = pb->next;
    } else {
      //指数相同,对应项的系数相加
      //若系数相加等于零,则此项相加后消失,pa 和 pb 一起前进一位
      if (pa->coef + pb->coef == 0) {
        qa = pa->next; qb = pb->next;
        free(pa);   free(pb);
        pa = qa;   pb = qb;
      } else {
        s = (ployn * )malloc(sizeof(ployn));
        //新结点 s 的系数为两个单项式的系数之和
        s->coef = pa->coef + pb->coef;
        s->exp = pa->exp;
        pc->next = s;
        pc = s;
        qa = pa->next; qb = pb->next;
        free(pa);   free(pb);
        pa = qa;   pb = qb;
      }
    }
  } //end_while
}
```

2.4.2　魔法师发牌问题

1. 问题描述

魔法师利用一副牌中的 13 张黑桃,预先将它们排列好后叠放在一起,牌面向下。对观众说:“我不看牌,只数数就可以猜到每张牌是什么,现在我大声地数,大家请看。”

魔法师将最上面的那张牌数为1,把它翻过来正好是黑桃A,将黑桃A放在桌子上。然后按顺序从上向下数手上的余牌,第二次数1、2,将第一张牌放在这些牌的下面,将第二张牌翻过来,正好是黑桃2,也将它放在桌子上;第三次数1、2、3,将前面两张一次放在这些牌的下面,再翻第三张牌正好是黑桃3。这样,依次进行,将13张牌全翻出来,准确无误。请问魔术师手中的牌原始顺序是怎样安排的?

2. 解题思路

对问题进行建模分析,首先建立长度为13的链表并初始化链表中的每个元素为0,表示链表中什么牌也不放。接下来根据魔法师发牌的顺序把13张牌存入链表中;例如,第一张牌存入第一个位置,第二张牌存入距离第一张牌两个空位的位置。需要注意的是,在存牌前只需记录当前链表中空位的个数并根据这个数字存放新牌。若一个位置已经被某张牌占据,则这个位置不可以再存放新牌。

3. 代码实现

```
#include "CirList.h"
#include <iostream>
using namespace std;
void main(int argc, char argv**) {
  CirList<int> poker;
  for (int i = 0; i < 13; i++)
    poker.AddTail(0);            //创建循环链表,存储13张扑克牌
  poker.SetBegin();
  poker.GetNextNode();
  for(i = 1; i < 14; i++) {
    poker.SetData(i);
    for(int j = 0; j <= i; j++) {
      poker.GetNextNode();   //寻找插牌位置
      //若当前位置已有牌,则顺序查找下一个位置
      if(poker.GetCur()->GetData() != 0) j--;
      if(u == 13) break;     //插牌完毕
    }
  }
  poker.SetBegin();
  poker.GetNextNode();
  for(i = 0; i < 13; i++) {
    cout << poker.GetCur()->GetData() << "*";
    poker.GetNextNode();
  }
  cout << endl;
}
```

2.4.3 约瑟夫问题

1. 问题描述

有n个人排成一圈,并给每个人编号1～n。现在从1号开始报数,若报数字m的人退出队伍,剩余的人从退出者下一个位置开始继续刚才的报数,直到整个队伍中只剩下一个人

为止。请问最后一个人的编号为几？

2. 解题思路

采用循环链表或队列均能解决问题。若采用循环队列时，报数的规律为1，2，…，m，报数1，2，…，m－1的人出队列并立即站到队列的队尾，数m的人出队列。报数过程反复进行，直至队列中只剩最后一人。若采用循环链表时，同样的报数规律，凡报数为1，2，…，m－1的人，均依次进行遍历，数m的人均从队伍中删除，过程反复进行，直至队列中只剩最后一人。

3. 代码实现

下面采用链表模拟约瑟夫问题。

```
#include "CirList.h"
#include <iostream>
using namespace std;
void main(int argc, char * * argv) {
  CirList<int> jos;                  //新建循环单链表,模拟约瑟夫问题
  int m; cin >> m;
  for(int i = 1; i <= n; i++)
    jos.AddTail(i);                  //向链表中添加1~n,代表编号为1~n的人
  jos.SetBegin();                    //从链表首结点开始报数
  int length = jos.GetCount();       //记录原始队伍中的人数
  for (i = 1; i < length; i++) {
    for (int j = 1; j < m; j++)
      jos.GetNext();                 //报数1~m-1的结点依次遍历
    jos.RemoveThis();                //报数m的结点删除(出队伍)
  }
  cout << jos.GetNext() << endl;     //输出链表中最后一个结点的编号
}
```

本章小结

本章主要学习了基本的线性结构线性表，主要学习要点如下。

- 掌握线性表的基本定义和相关概念。
- 理解线性表的逻辑结构特性和基本运算。
- 理解线性表的两种存储方法，即顺序表和链表的类型定义。
- 掌握顺序表和链表上各种基本运算的实现和应用。
- 综合运用线性表解决一些较复杂的实际问题。

第3章 栈与队列

栈和队列是两种特殊的线性结构。其特殊性在于栈和队列的插入、删除受限制,因此栈和队列也称为操作受限的线性表。从数据类型的角度看,它们是和线性表大不相同的两类重要的抽象数据类型。

本章将讨论栈和队列的基本概念、存储结构、基本操作以及具体应用的实现。

3.1 栈



视频讲解

3.1.1 栈的概述

栈(stack)是一种限定只能在一端进行插入或删除操作的线性表。表中允许进行插入、删除操作的一端称为**栈顶**(top),表的另一端称为**栈底**(bottom)。不含元素的空表称为**空栈**。

栈的主要特点是**"先进后出"**,即先入栈的元素后出栈。每次进栈的数据元素都放在当前栈顶元素之前成为新的栈顶元素,每次出栈的数据元素都是当前栈顶元素。栈也称为**先进后出表**。例如,家里的碗通常在洗干净后一个一个地摞在一起,使用时一个一个拿,最先被拿的一定是最上面的碗,而最后被拿的是最下面的碗。这就是典型的栈"先进后出"原理。

假设栈 S=(a_1,a_2,…,a_n),则称 a_1 为栈底元素,a_n 为栈顶元素。栈中元素按 a_1,a_2,…,a_n 的次序进栈,退栈的第一个元素应为栈顶元素。换句话说,栈的修改是按先进后出的原则进行的,如图 3.1 所示。

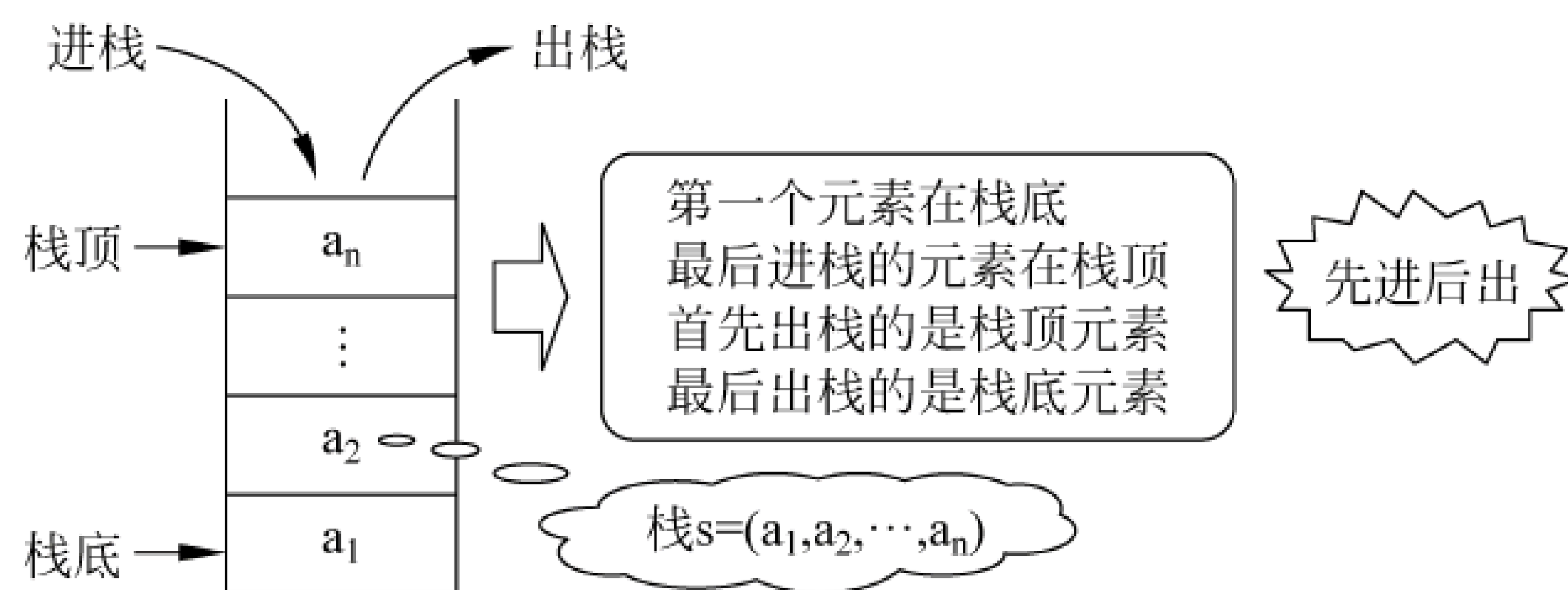


图 3.1 栈的示意图

栈的基本操作除了在栈顶进行插入或删除外,还有栈的初始化、取栈顶元素等。下面给出栈的抽象数据类型的定义。

```
ADT Stack
{
```

数据对象:D={a_i| 1≤i≤n,a_i 为 elemtype 类型}
数据关系:R={<a_i,a_{i+1}>|a_i,a_{i+1}∈D,i=1,2,…, n−1}
基本操作:
InitStack(&S):初始化栈 S.
  操作结果:返回初始化空栈 S.
DestroyStack(&S):销毁栈 S.
  初始条件:栈 S 已知存在.
  操作结果:释放栈 S 占用的存储空间
StackEmpty(S):判断栈 S 是否为空.
  初始条件:栈 S 已知存在.
  操作结果:若栈 S 为空,则返回 true;否则返回 false.
Push(&S,e):元素进栈.
  初始条件:栈 S 已知存在,新元素为 e.
  操作结果:将元素 e 插入到栈 S 中作为栈顶元素.
Pop(&S,&e):元素出栈.
  初始条件:栈 S 已知存在.
  操作结果:从栈 S 中退出栈顶元素,并将其值赋给变量 e.
GetTop(S,&e):取栈顶元素.
  初始条件:栈 S 已知存在.
  操作结果:返回当前的栈顶元素,并将其值赋给 e.
} ADT Stack

【例 3.1】 若元素的进栈顺序为 12345,能否得到 31425 的出栈顺序?

解:为了让 3 作为第一个出栈元素,1、2 先进栈,此时要么 2 出栈,要么 4 进栈后出栈或其他进出栈情况,但出栈的第 2 个元素不可能是 1,因为 2 压在 1 的上方,所以得不到 31425 的出栈顺序。

【例 3.2】 用 S 表示进栈操作,X 表示出栈操作,若元素的进栈顺序为 1234,为了得到 1342 的出栈顺序,请给出相应的 S 和 X 操作串。

解:为了得到 1342 的出栈顺序,其操作过程是 1 进栈,1 出栈,2 进栈,3 进栈,3 出栈,4 进栈,4 出栈,2 出栈。因此,相应的 S 和 X 操作串为 SXSSXSXX。

【例 3.3】 对于一个栈,给出输入项 A、B、C,如果输入项序列由 ABC 组成,试给出所有可能的输出序列。

解:A 进 A 出 B 进 B 出 C 进 C 出⇒ABC
A 进 A 出 B 进 C 进 C 出 B 出⇒ACB
A 进 B 进 B 出 A 出 C 进 C 出⇒BAC
A 进 B 进 B 出 C 进 C 出 A 出⇒BCA
A 进 B 进 C 进 C 出 B 出 A 出⇒CBA

注意:根据栈的“先进后出”原理,A 进栈,B 进栈,C 进栈,C 出栈,第二个出栈的不可能是 A,只能是 B,因此不会产生输出序列 CAB。

注意:n 个元素组成的序列入栈,其出栈序列共有$\frac{1}{n+1}C_{2n}^{n}$种。


视频讲解

3.1.2 栈的顺序存储结构

由于栈从组成元素的逻辑关系上看是线性结构,因此可以像线性表一样采用顺序存储结构,即分配一组地址连续的存储单元依次存放自栈底到栈顶的数据元素,同时设指针 top

指示栈顶元素在顺序栈中的位置。采用顺序存储结构的栈称为**顺序栈**。

假设栈的元素最大个数不超过正整数 MaxSize,所有的元素都具有同一数据类型(即 elemtype),则可用下列方式定义顺序栈类型 SqStack。

```
typedef struct
{
    elemtype data[MaxSize];      //存储栈中的数据元素
    int top;                     //栈顶指针,指示栈顶位置
} SqStack;                       //顺序栈类型
```

注意:SqStack 顺序栈的本质即为顺序表,但却有自己的独特性(只允许在栈顶位置插入和删除),为了突出栈顶位置,需要定义一个成员 top 指示栈顶位置;算法中往往采用定义指针 * top 方式指示位置,但从上面类型定义中看出 top 被说明为 int 类型,主要是考虑到 data[ ]数组的使用习惯,数组可以通过不同的下标区别不同存储空间,因此 int top 在此表示指示栈顶位置的下标,同时,因为有了 top 栈顶下标,还可以计算出顺序栈的长度(本教材中,top+1 的值刚好等于表长),这样就降低了顺序栈的运算难度。

根据上述顺序栈的类型定义,顺序栈 S 可定义成如下两种形式。

```
SqStack  * S; 或 SqStack S;
```

栈到顺序栈的映射如图 3.2 所示。本节采用栈指针 S(不同于栈顶指针 top)的方式建立和使用顺序栈,如图 3.3 所示。

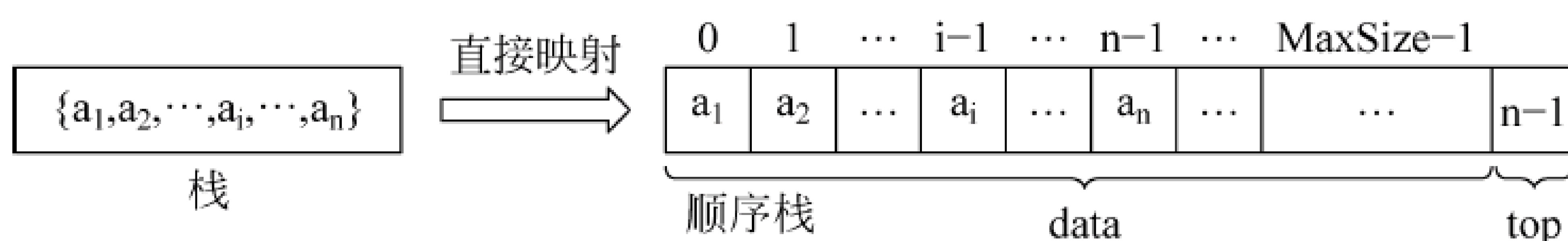


图 3.2 栈到顺序栈的映射



图 3.3 顺序栈指针 S

操作受限制的顺序栈没有顺序表操作灵活,运算较为简单,关于顺序栈是如何进行操作的可参照图 3.4。

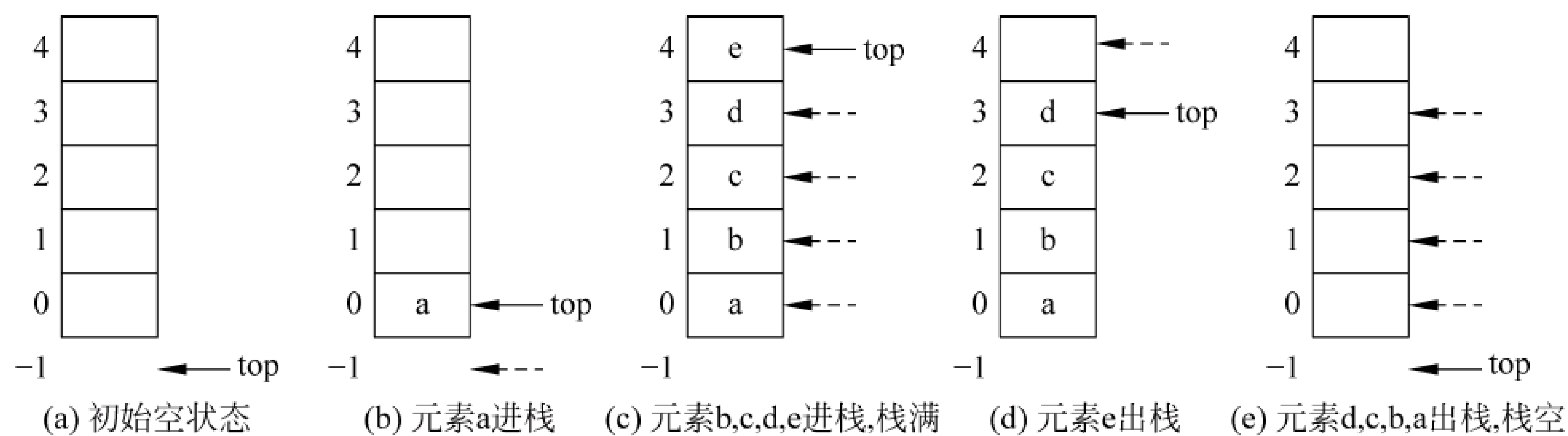


图 3.4 顺序栈操作示意图

对于顺序栈 S:

栈空的条件是 **S→top==-1 或 S→top+1==0**,此刻无法进行出栈运算,否则产生“下溢”。

栈满的条件是 **S→top==MaxSize-1**，此刻无法进行进栈运算，否则产生“下溢”。

在条件允许的情况下进行一个元素的进栈，需要先将 top 下标后移一位（S→top++），然后将元素压入栈顶位置；弹出栈顶元素时，先弹出栈顶元素，然后再将 top 下标前移一位（S→top--）。具体算法实现如下。

1. 栈的初始化

初始化顺序栈空间，并初始化栈顶指针的位置，即将栈顶指针指向-1 即可。

```
void InitStack(SqStack * &S)
{
S=(SqStack * )malloc(sizeof(SqStack));
S→top=-1;
}
```

初始化完成后，栈 S 为空栈，算法的时间复杂度为 O(1)。

2. 销毁栈

释放顺序栈 S 占用的存储空间。

```
void DestroyStack(SqStack * &S)
{
    free(S);
}
```

算法的时间复杂度为 O(1)。

3. 判断栈是否为空

顺序栈 S 为空的条件是 S→top==-1；

```
bool StackEmpty(SqStack * S)
{
    return(S→top == -1);
}
```

算法的时间复杂度为 O(1)。

4. 进栈

在栈未满的条件下，先将栈顶指针增加 1，然后在栈顶指针 top 指向位置插入元素 e。

```
bool Push(SqStack * &S, elemtype e)
{ if (S→top==MaxSize-1)          //栈满元素无法进栈
      return false;
    S→top++;                      //栈顶指针增 1
    S→data[S→top]=e;              //元素 e 放在栈顶指针处
    return true;
}
```

算法的时间复杂度为 O(1)。

5. 出栈

在栈不为空的条件下，先将栈顶元素赋给变量 e，然后将栈顶指针减 1。

```
bool Pop(SqStack *&S, elemtype &e)
{
if(S->top==-1)              //栈为空，无法出栈
    return false;
e=S->data[S->top];          //变量 e 接收栈顶元素
S->top--;                   //栈顶指针减 1
return true;
}
```

算法的时间复杂度为 O(1)。

6. 读取栈顶元素

在栈不为空的条件下，将栈顶元素赋给变量 e。

```
bool GetTop(SqStack *S, elemtype &e)
{
    if(S->top==-1)
      return false;
    e=S->data[S->top];
return true;
}
```

算法的时间复杂度为 O(1)。

【例 3.4】 编写一个算法，利用顺序栈判断一个字母串是否是对称串。对称串是指从左向右读取和从右向左读取的序列相同。

分析：对于字符串 str，先将其所有元素依次进栈；然后从头开始扫描字符串 str，并取出栈元素，将两者进行比较，若不相同，则返回 false。当 str 扫描完毕后，返回 true。实际上，从头开始扫描 str 是从左向右读字符串，出栈序列则是从右向左读字符串，若两者相等，则说明该串是对称串。算法如下。

```
bool symmetry(elemtype str[])           //判断 str 是否为对称串
{
   int i, elemtype e;
   SqStack *st;                         //定义顺序栈 st
   InitStack(st);                       //初始化栈
   for(i=0;str[i]!='\0';i++)            //将 str 的所有元素进栈
      Push(st, str[i]);
   for(i=0;str[i]!='\0';i++)            //处理 str 的所有字符
   { Pop(st, e);                        //退栈元素 e
         if(str[i]!=e)                  //若 e 与当前字符串不同，则表示不是对称串
         { DestroyStack(st);            //销毁栈
               return false;            //返回 false
         }
```

```
    }
    DestroyStack(st);                //销毁栈
    return true;                     //返回 true
}
```

3.1.3 栈的链式存储结构



采用链式存储的栈称为**链栈**，本质是带头结点的单链表。链栈的优点是不存在栈满上溢的情况。规定栈的所有操作都在单链表的表头进行，如图 3.5 所示是头结点为 S 的链栈，第一个数据结点是栈顶结点，最后一个结点是栈底结点。栈中元素自栈底到栈顶依次是 $a_1, a_2, \cdots, a_n$。

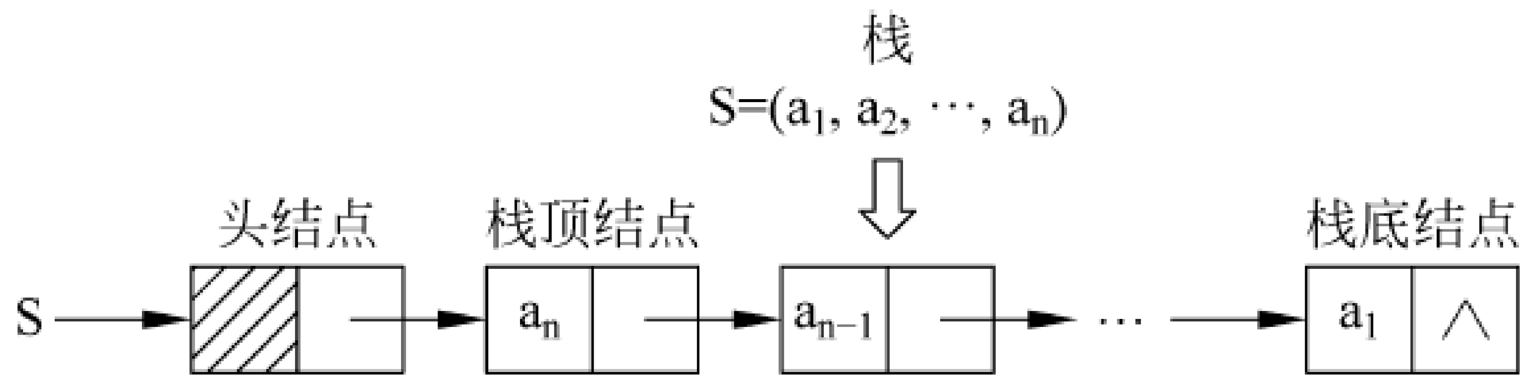


图 3.5　链栈 S 结构示意图

链栈中数据结点的类型定义如下。

```
typedef struct LNode             //链栈结点 LNode 类型
{
    elemtype data;               //数据域
    struct LNode * next;         //指针域
} * ListStack;                   //链栈类型
```

在以 S 为头结点的链栈中：

- 栈空的条件为 S->next==NULL；
- 由于只有在内存溢出时才会出现栈满，而通常不考虑内存溢出的情况，所以在链栈中可以人为地不存在栈满的情况；
- 结点 p 进栈的操作是在头结点 S 之后插入结点 p；出栈的操作是取出头结点 S 之后的结点的 data 值并将该结点删除。对应栈的基本运算算法如下。

1. 初始化栈

```
void InitStack(ListStack &S)
{
    S =(ListStack)malloc(sizeof(ListStack));
    S->next=NULL;
}
```

2. 销毁栈

释放栈 S 占用的全部存储空间。

```
void DestroyStack(ListStack &S)
{
```

```
    LNode *p=S, *q=S->next;           //p指向头结点,q指向首结点
    while(q!=NULL)                    //循环到q为空
    {
    free(p);                          //释放p结点
    p=q;
    q=p->next;
    }
      free(p);                        //此时p指向尾结点,释放其空间
}
```

3. 判断栈是否为空

栈S为空的条件是S->next==NULL,即单链表中没有数据结点。

```
bool StackEmpty(ListStack S)
{
    return(S->next==NULL)
}
```

4. 进栈

将新数据结点插入头结点之后。

```
void Push(ListStack &S, elemtype e)
{
    LNode *p;
    p=(LNode *)malloc(sizeof(LNode));          //生成新结点p
    p->next=e;
    p->next=S->next;                           //将p结点插入作为首结点
    S->next=p;
}
```

5. 出栈

在栈不为空的条件下,将头结点的指针域所指数据结点的数据域赋给e,然后将该数据的结点删除。

```
bool Pop(ListStack &S, elemtype &e)
{
    LNode *p;
    if(S->next==NULL)             //栈空的情况
        return false;
    p=S->next;                    //p指向首结点
    e=p->data;                    //提取首结点值
    S->next=p->next;              //删除首结点
    free(p);                      //释放被删结点的存储空间
    return true;
}
```

6. 取栈顶元素

在栈不为空的条件下，将头结点的指针域所指数据结点的数据域赋给 e。

```
bool GetTop(ListStack S, elemtype &e)
{
    if(S→next==NULL)           //栈空的情况
        return false;
    e=S→next→data;             //提取首结点值
    return true;
}
```

【例 3.5】 一个链栈 S 初始为空，将元素 a，b，d，c 依次入栈，请分别画出将元素入栈的结构示意图。

分析：首先画出空链栈 S，头指针 S 也是栈顶指针，分别将元素 a，b，d，c 生成新结点，插到头结点的后面，可参照首插法创建链表画法。链栈 S 的入栈过程如图 3.6 所示。

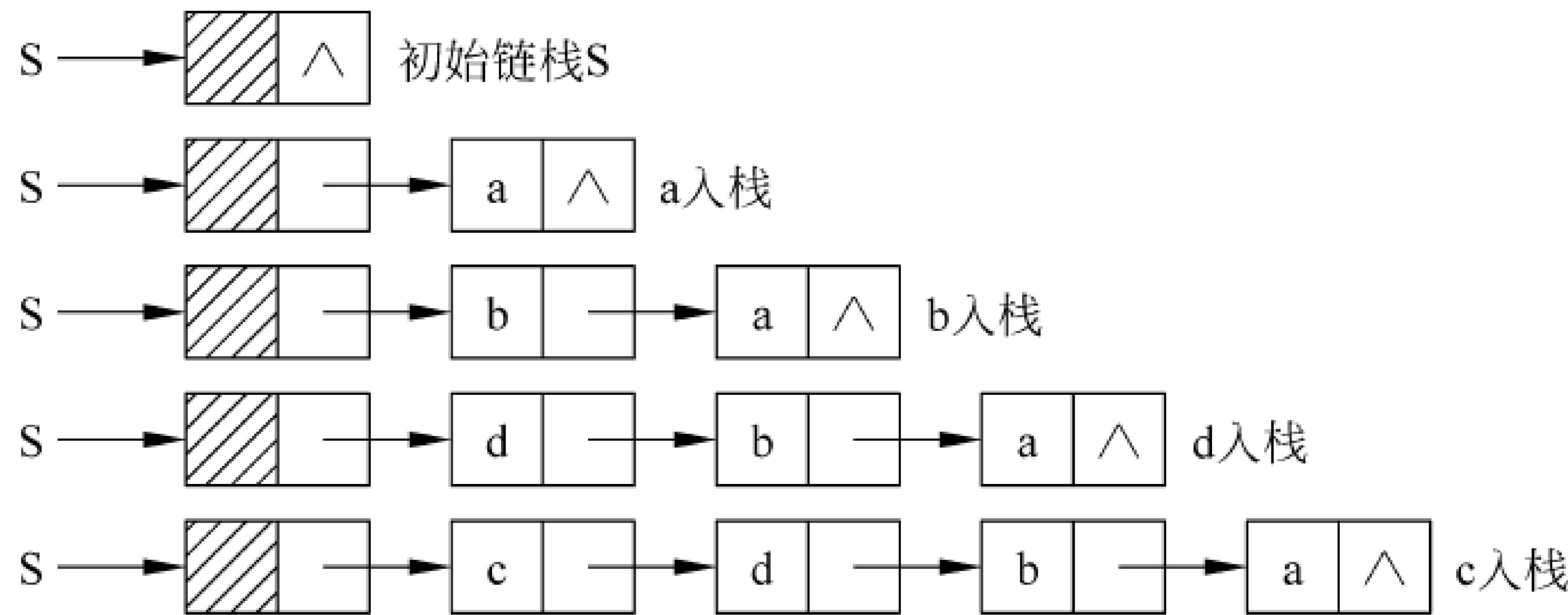


图 3.6 链栈 S 的入栈过程

3.2 栈综合案例

栈结构具有先进后出的固有特性，致使栈成为程序设计中的重要工具。换言之，栈的应用可被认为是在进栈和出栈的过程中对栈特性的应用。本节将讨论几个顺序栈应用的典型示例。

3.2.1 进制转换



视频讲解

十进制数 N 和其他 R 进制数的转换是计算机实现计算的基本问题，其解决方法有很多，其中的一个简单算法基于下列原理：

N=(N div R) * R+N mod R(其中，div 为整除运算，mod 为求余运算)

例如，$(1357)_{10}$转换为$(2515)_8$。转换过程如图 3.7 所示。

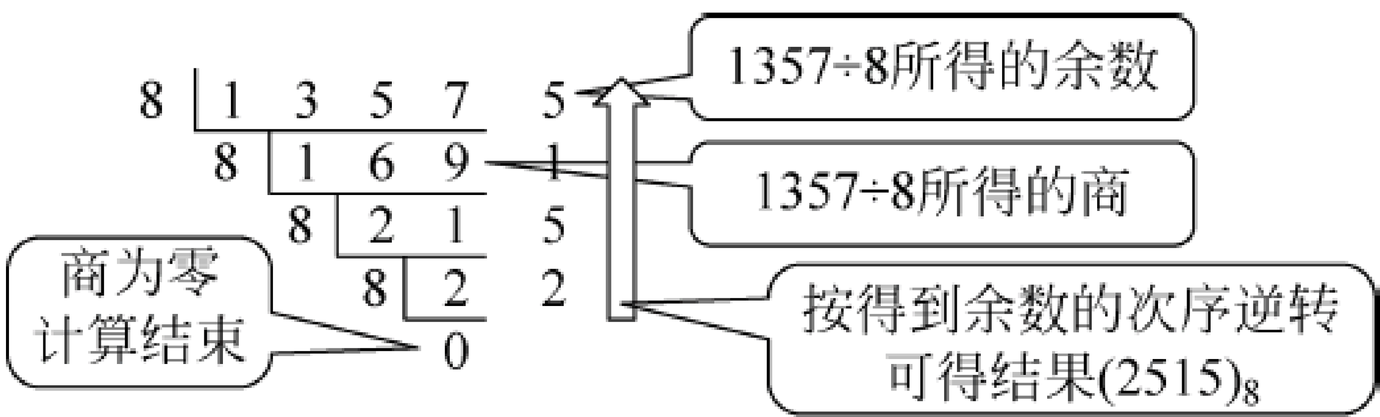


图 3.7 十进制转八进制

假设编制一个满足下列要求的程序：对于输入的任意一个非负十进制整数，打印输出与其等值的八进制数。由于上述计算过程是从低位到高位顺序产生八进制数的各个数位，而打印输出一般来说应从高位到低位进行，恰好与计算过程相反。因此，若将计算过程中得到的八进制数的各位顺序进栈，则按出栈序列打印输出的序列即为与输入对应的八进制数。

```
void conversion()
{
 //对于输入的任意一个非负十进制整数,打印输出与其等值的八进制数
 InitStack(S);          //构造空栈
 scanf("%d" , &N);
 while (N)
 { Push(S, N%8);
  N=N/8;
 }
 while(!StackEmpty(S))
{
  Pop(S,e);
  printf("% d",e);
 }
}
```

这是利用栈的先进后出特性的最简单的例子。在这个例子中，栈操作的序列是直线式的，即先全部入栈，然后全部出栈。也许，有的读者会提出疑问：用数组直接实现不也很简单吗？仔细分析上述算法可以看出，栈的引入简化了程序设计的问题，划分了不同的关注层次，使思考范围缩小了。而用数组不仅掩盖了问题的本质，还要分散精力去考虑数组下标增减等细节问题。

注意：将十进制数转化成 R 进制数的计算过程如何呢？上面的算法实现过程可适用于十进制转二进制、十进制转八进制，但对十进制转十六进制需要稍做修改，因为十六进制的数码为 0～9 及 A～F，因此将余数从栈 S 中弹出时，需要对余数进行讨论，根据讨论结果做不同的输出，修改算法如下。

```
while(!StackEmpty(S))
{
  Pop(S,e);
  printf("% d",e);
 }//输出余数部分修改为
while(!StackEmpty(S))
{
  Pop(S,e);
  if(e<10)
  printf("% c",e+48);
  else
  printf("%c",e+55);
}
```

3.2.2 表达式求值

表达式求值是程序设计语言编译中的一个最基本问题。它的实现是栈应用的又一个典型例子。这里介绍一种简单直观、广为使用的算法，通常称为“算符优先法”。

要把一个表达式翻译成能正确求值的一个机器指令序列，或者直接对表达式求值，首先要能够正确解释表达式。例如，要对下面的算术表达式求值：

$$4+5\times2-20/5$$

首先要了解四则算术运算的规则，即

step1：先乘除，后加减。

step2：从左算到右。

step3：先括号内，后括号外。

由此，这个算术表达式的计算顺序应为

$$4+5\times2-20/5=4+10-20/5=4+10-4=14-4=10$$

计算简单的算法表达式，常用的方法是“**算符优先法**”，就是根据这个运算优先关系的规定实现对表达式的编译或解释执行的。任何一个表达式都是由操作数（operand）、运算符（operator）和界限符（delimiter）构成的，我们称它们为单词。一般地，操作数既可以是常数，也可以被说明为变量或常量标识符；运算符可以分为算术运算符、关系运算符和逻辑运算符 3 类；基本界限符有左右括号和表达式结束符等。为了叙述的简洁，我们仅讨论简单算术表达式的求值问题。每种表达式只含加、减、乘、除 4 种运算符。不难将它推广到更一般的表达式上。我们把运算符和界限符统称为算符，它们构成的集合命名为 OP。根据上述 3 条运算规则，在运算的每一步中，任意两个相继出现的算符 e1 和 e2 之间的优先关系至多是下面 3 种关系之一。

- e1＜e2：e1 的优先权低于 e2。
- e1＝e2：e1 的优先权等于 e2。
- e1＞e2：e1 的优先权高于 e2。

通过表 3.1，观察基本算符之间的有限关系。

表 3.1　算符间的优先关系

运　算　符	+	−	*	/	(	)	#
+	＞	＞	＜	＜	＜	＞	＞
−	＞	＞	＜	＜	＜	＞	＞
*	＞	＞	＞	＞	＜	＞	＞
/	＞	＞	＞	＞	＜	＞	＞
(	＜	＜	＜	＜	＜	＝	
)	＞	＞	＞	＞		＞	＞
#	＜	＜	＜	＜	＜		＝

由规则 **step3** 可知，+、−、* 和/为 e1 时的优先性，均低于“(”但高于“)”，由规则 **step2** 可知，当 e1＝e2 时，令 e1＞e2，“#”**是表达式的结束符**。为了算法简洁，在表达式的最左边也虚设一个“#”构成整个表达式的一对括号。表中的“(”＝“)”表示当左右括号相遇时，括

号内的运算已经完成。同理,“#”=“#”表示整个表达式求值完毕。“)”与“(”“#”与“)”以及“(”与“#”之间无优先关系,这是因为表达式中不允许它们相继出现,一旦遇到这种情况,则可以认为出现了语法错误。在下面的讨论中,我们暂假定所输入的表达式不会出现语法错误。

为实现算符优先算法,可以使用两个工作栈:一个称为 OPTR,用以寄存运算符;另一个称为 OPND,用以寄存操作数或运算结果。算法的基本思想如下。

step1:置操作数栈为空栈,表达式起始符“#”为运算符栈的栈底元素。

step2:依次读入表达式中的每个字符,若是操作数,则进 OPND 栈;若是运算符,则和 OPTR 栈的栈顶运算符比较优先权后做出相应操作,直至整个表达式求值完毕(即 OPTR 栈的栈顶元素和当前读入的字符均为“#")。

```
OperandType EvaluateExpression()
{
//算术表达式求值的算符优先算法.设 OPTR 和 OPND 分别为运算符栈和运算数栈
InitStack(OPTR);
Push(OPTR,'#');
InitStack(OPND);
c=getchar();
while (c!='#'|| GetTop(OPTR)!='#')
{
 if(!In(c, OP))
{
 Push(OPND, c);
 c=getchar();
}
 else
  switch (Precede(GetTop(OPTR),c))
{
  case'<':         //栈顶元素优先权低
    Push(OPTR, c);
    c=qetchar();
    break;
  case '=':        //脱括号并接收下一字符
    Pop(OPTR,x);
    c=getchar();
    break;
  case'>':         //退栈并将运算结果入栈
    Pop(OPTR, theta);
    Pop(OPND, b);
    Pop(OPND, a);
    Push(OPND, Operate(a, theta, b));
    break;
  }
 }
 return GetTop(OPND);
}
```

算法中还调用了两个函数。其中，Precede 是判定运算符栈的栈顶运算符 e1 与读入的运算符 e2 之间优先关系的函数；Operate(a，theta，b)为进行二元运算的函数，即将操作数据 a,b 进行 theta 运算，如果是编译表达式，则产生这个运算的一组相应指令并返回存放结果的中间变量名；如果是解释执行表达式，则直接进行该运算，并返回运算的结果。

例如，利用算法 EvaluateExpression 对算术表达式 3＊(2＋2)/6 求值，操作过程见表 3.2。

表 3.2　对表达式 3＊(2＋2)/6 求值的过程

步　骤	OPTR 栈	OPND 栈	输 入 字 符	主 要 操 作
1	#		3＊(2＋2)/6#	PUSH(OPND,'3')
2	#	3	＊(2＋2)/6#	PUSH(OPTR,'＊')
3	#＊	3	(2＋2)/6#	PUSH(OPTR,'(')
4	#＊(	3	2＋2)/6#	PUSH(OPND,'2')
5	#＊(	3 2	＋2)/6#	PUSH(OPTR,'＋')
6	#＊(＋	3 2	2)/6#	PUSH(OPND,'2')
7	#＊(＋	3 2 2	)/6#	OPERATE('2','＋','2') PUSH(OPND, '4')
8	#＊(	3 4	)/6#	POP(OPTR)
9	#＊	3 4	/6#	OPERATE('3','＊','4') PUSH(OPND, '12')
10	#	12	/6#	PUSH(OPTR, '/')
11	#/	12	6#	PUSH(OPND, '6')
12	#/	12 6	#	POP(OPTR)
13	#	2	#	OPERATE('12','/','6') PUSH(OPND,'2')
14	#	2	#	RETURNGETTOP(OPND)

3.2.3　检验表达式中的括号匹配情况

假设在一个算术表达式中可以包含 3 种括号：圆括号“(”和“)”、方括号“[”和“]”和花括号“{”和“}”，并且这 3 种括号可以按任意的次序嵌套使用。例如，[…{…}…[…]…]…[…]…(…)…。现在需要设计一个算法，用来检验在输入的算术表达式中所使用括号的合法性。

解：算术表达式中各种括号的使用规则为

出现左括号，必有相应的右括号与之匹配，并且每对括号之间可以嵌套，但不能出现交叉情况。

可以利用一个栈结构保存每个出现的左括号，当遇到右括号时，从栈中弹出左括号，检验匹配情况。在检验过程中，若遇到以下几种情况之一，就可以得出括号不匹配的结论。

step1：当遇到某一个右括号时，栈已空，说明到目前为止，右括号多于左括号。

step2：从栈中弹出的左括号与当前检验的右括号类型不同，说明出现了括号交叉情况。

step3：算术表达式输入完毕，但栈中还有没有匹配的左括号，说明左括号多于右括号。

下面是解决这个问题的完整算法。

```
typedef char elemtype;
bool Check()
{
 char ch;
 InitStack(&S);                                      //初始化栈 S
 while((ch=getchar())!='\n')                         //以字符序列的形式输入表达式
 { switch(ch)
  {
   case(ch=='('||ch=='['||ch=='{'):Push(&S,ch);break;//遇左括号入栈
   case(ch==')'):                                    //遇右括号时,检测匹配情况
     if(StackEmpty(S)) return false;
     else{ Pop(&S,&ch);
        if(ch!='(') return false; }
     break;
   case(ch==']'):
     if(StackEmpty(S)) return false;
     else{ Pop(&S,&ch);
        if(ch!='[') return false; }
     break;
   case(ch=='}'):
     if(StackEmpty(S)) return false;
     else{ Pop(&S,&ch);
        if(ch!='{') return false; }
     break;
   default:break;
  }
 }
 if(StackEmpty(S)) return true;
 else return false;
}
```

3.2.4 栈与递归问题

栈还有一个重要的应用是在程序设计中实现递归与非递归算法的转换。递归在算法设计中是一种重要的处理手段，在计算方法、数据建模、行为策略等研究中有广泛的应用。

所谓递归，是指若在一个函数过程内部直接或间接地出现定义本身的应用，就称它是递归的，或者是递归定义的。递归可以分为**直接递归**和**间接递归**两种。两种递归的调用形式如下。

```
函数值类型 A(形参列表)
{
 ……
 A(实参列表)
 ……
}
```

过程 A 在结束执行前直接调用了过程 A 本身，则称为直接递归调用。

```
函数值类型 A(形参列表)
{
 ……
 B(实参列表)
 ……
}
函数值类型 B(形参列表)
{
 ……
 A(实参列表)
 ……
}
```

过程 A 调用了过程 B,而过程 B 又调用了过程 A,即过程 A 通过过程 B 调用了自身,则称为间接递归调用。

递归算法比非递归算法更容易被设计,尤其是当问题本身或所设计的数据结构是递归定义的时候,使用递归算法非常合适。采用递归一般有 3 种情况。

1. 定义是递归的

如数学中斐波那契数列的定义:

$$f(n)=\begin{cases} n & n=0 \text{ 或 } n=1 \\ f(n-2)+f(n-1) & n\geqslant 2 \end{cases}$$

其产生的序列为 0,1,1,2,3,5,8,13,21,…。

其递归算法如下。

```
long f(int n)
{
 if(n<2)
   return n;
 else
   return f(n-2)+f(n-1);
}
```

2. 数据结构是递归的

如树、二叉树、广义表、链表等,由于结构固有的递归性,所以它的操作可递归地描述。例如,链表结点 LNode 的定义。

```
typedef struct LNode{
 elemtype data;
 struct LNode  * next;      //指针 next 指示 LNode 型结点
};
```

例如,一个不带头结点的非空链表 L,若结点的数据值均为整数(int 型),则使用递归算法计算所有结点的数据值之和。

```
int sum(LinkList L)
{
 if(L)
    return (L->data+sum(L->next));
 else
    return 0;
}
```

3. 问题解法是递归的

如八皇后问题、汉诺塔问题等。这类问题本身没有明显的递归结构，但是通过对问题求解过程分析发现，使用递归求解比迭代求解更简单。

例如，解决汉诺塔问题的过程，递归方法就是一种好的方法。n阶汉诺塔问题中，如果用函数hanoi(n,X,Y,Z)表示把n个盘子从X移动到Z，中间借用Y作为临时中转站，用函数move(n,X,Y)表示把第n个盘子从X移动到Y的过程，则解决汉诺塔问题的过程可描述为如下算法。

```
void hanoi(int n, elemtype X,elemtype Y,elemtype Z)
{
 if(n==1)
    move(1,X,Z);             //X上唯一的盘子直接移动到Z上
 else
 {
    hanoi(n-1,X,Z,Y);        //X上前n-1个盘子通过Z移动到Y上
    move(n,X,Z);             //X上第n个盘子直接移动到Z上
    hanoi(n-1,Y,X,Z);        //Y上n-1个盘子通过X移动到Z上
 }
}
```

很明显，递归算法的优点是结构清晰、程序易读，但其缺点是时间效率低、空间消耗大、算法不容易优化。对于频繁使用的算法，常常需要将递归算法转换成非递归算法实现，利用栈可以将任何递归函数转化成非递归函数，其步骤如下。

1）入栈处理

step1：用一个工作栈替代递归函数中的栈，栈中的每个记录包括函数的所有参数：函数名、局部变量和返回地址。

step2：把所有递归调用语句改写成形参、局部变量和返回地址入栈的语句。

step3：修改确定本次递归调用时的实际参数的新值。

step4：转到函数的第一个语句。

2）出栈处理

step1：若栈空，算法结束，执行正常返回。

step2：若栈不空，从栈中退出参变量赋值给原来入栈时对应的参变量，并退出返回地址。

step3：转到执行返回地址处的语句继续执行。

通过以上步骤，可将任何递归算法改写成非递归算法。

3.3 队　　列



3.3.1 队列的定义和抽象数据类型

队列(queue)是一种先进先出(First In First Out,**FIFO**)的线性表。它只允许在表的一端插入元素,而在另一端删除元素。这和我们日常生活中的排队是一致的,最早进入队列的元素最早离开。在队列中,允许删除的一端称为**队头**(front),允许插入的一端称为**队尾**。向队列中插入元素称为**入队列**或**进队列**,而删除队列中的元素称为**出队列**或**离队列**。

假设线性表 $Q=(a_1,a_2,\cdots,a_n)$为队列,若 a_1是队首元素, a_n则是队尾元素。队列中的元素是按照 $a_1,a_2,\cdots,a_n$的次序进入的,退出队列也只能按照这个次序依次退出。也就是说,只有在 $a_1,a_2,\cdots,a_{n-1}$都离开队列之后,a_n才能退出队列。图 3.8 是队列示意图。

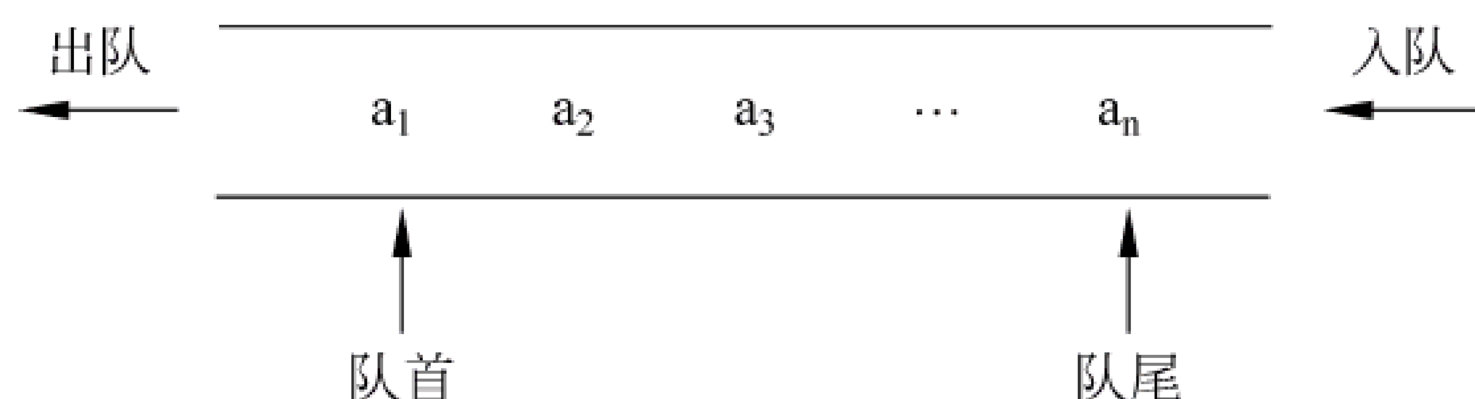


图 3.8　队列示意图

队列的例子在生活中非常常见,相比栈而言,队列在处理元素时比较“公平”,能体现出先来先服务的原则。例如,到医院看病,首先需要到挂号处挂号,然后按号码顺序就诊。又如,乘坐公共汽车应在车站排队,车来后按顺序上车。再如,在 Windows 这类多任务的操作系统环境中,每个应用程序响应一系列的“消息”,像用户单击鼠标;拖动窗口这些操作都会导致向应用程序发送消息。为此,系统将为每个应用程序创建一个队列,用来存放发送给该应用程序的所有消息,应用程序的处理过程就是不断地从队列中读取消息,并依次给予响应。

队列的操作与栈的操作类似,也有以下几种。不同的是,队列的删除是在表的头部(即队头)进行,而栈的删除只能在表尾(即队尾)进行。

下面给出队列的抽象数据类型定义。

```
ADT Queue
{
 数据对象:D={ai| 1≤i≤n,ai 为 elemtype 类型}
 数据关系:R={<ai,ai+1>|ai,ai+1∈D,i=1,2, … ,n-1}
 基本操作:
 InitQueue(&Q):构造一个空队列 Q。
     操作结果:初始化空队列 Q。
 DestroyQueue(&Q):销毁队列。
     初始条件:队列 Q 已存在。
     操作结果:释放队列 Q 占用的存储空间。
 QueueEmpty(Q):判断队列是否为空。
     初始条件:队列 Q 已存在。
     操作结果:若队列 Q 为空,返回 true;否则返回 false。
 QueueLength(Q):返回 Q 的元素个数。
```

初始条件:队列 Q 已存在。
操作结果:返回队列的长度。
EnQueue(&Q, e):插入元素 e。
初始条件:队列 Q 已存在。
操作结果:插入元素 e,使之成为 Q 的新的队尾元素。
DeQueue(&Q, &e):删除 Q 的队首元素。
初始条件:队列 Q 已存在。
操作结果:删除 Q 的队首元素,并用 e 返回其值。
}ADT Queue

【例 3.6】 若元素的进队顺序为 12345,能否得到 54321 的出队顺序?

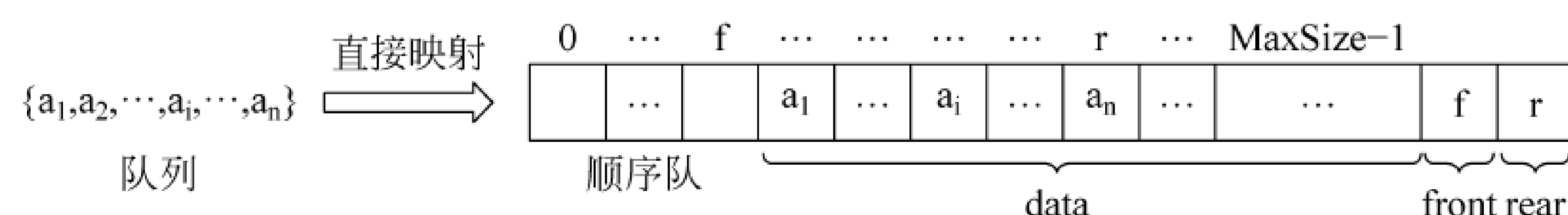
解:进队顺序为 12345,则出队顺序只有一种,即 12345(先进先出),所以不能得到 54321 的出队顺序。

3.3.2 队列的顺序存储

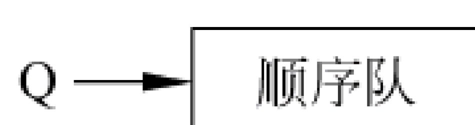

视频讲解

队列的本质是线性表,所以它可以像线性表一样采用顺序存储结构存储,即分配一块连续的存储空间存储队列中的元素,并用两个整型变量即队首指针(队头指针)和队尾指针分别存储队首元素和队尾元素的下标位置。采用顺序存储结构的队列称为**顺序队列**。

顺序队列存储结构如图 3.9(a)所示。本节采用队列指针 Q 的方式建立和使用顺序队列,如图 3.9(b)所示。



(a) 顺序队列存储结构



(b) 顺序队列指针Q

图 3.9 顺序队列

假设队列的元素个数不超过整数 MaxSize,所有元素都具有同一数据类型 elemtype,则顺序队列类型 SqQueue 的定义如下。

```
typedef struct
{
elemtype data[MaxSize];        //存放队列中的元素
int front, rear;               //队头和队尾指针
}SqQueue;                      //顺序队列类型
```

注意:SqQueue 顺序队列本质上为顺序表,但却有自己的独特性,只允许在队尾位置插入,队头位置删除,因此定义一个成员 rear 指示队尾位置,定义一个成员 front 指示队头位置;参照顺序栈 top 成员的定义方法,rear、front 分别表示指示队尾位置和队头位置的下标。同时,顺序队列的元素个数可以通过 rear 与 front 计算。通常,顺序队列 Q 可定义成如下两种形式。

```
SqQueue * Q; 或 SqQueue Q;
```

从图 3.10 可以看出，在顺序队列 Q 中：

队空的条件为 Q→rear-Q→front==0 或 Q→rear==Q→front。

队满的条件为 Q→rear-Q→front==MaxSize 或 Q→rear==MaxSize-1。

如图 3.10(b)所示，元素进队的操作是先将队尾指针 rear 增 1，然后将元素放在队尾处。

如图 3.10(d)所示，出队操作是先将队头指针 front 增 1，然后取出队头处的元素。

队尾指针总是指向当前队列中队尾的元素，而队头指针总是指向当前队列中队首元素的前一个位置；值得注意的是，在图 3.10(d)中，此刻入队列会出现“假溢出”现象。

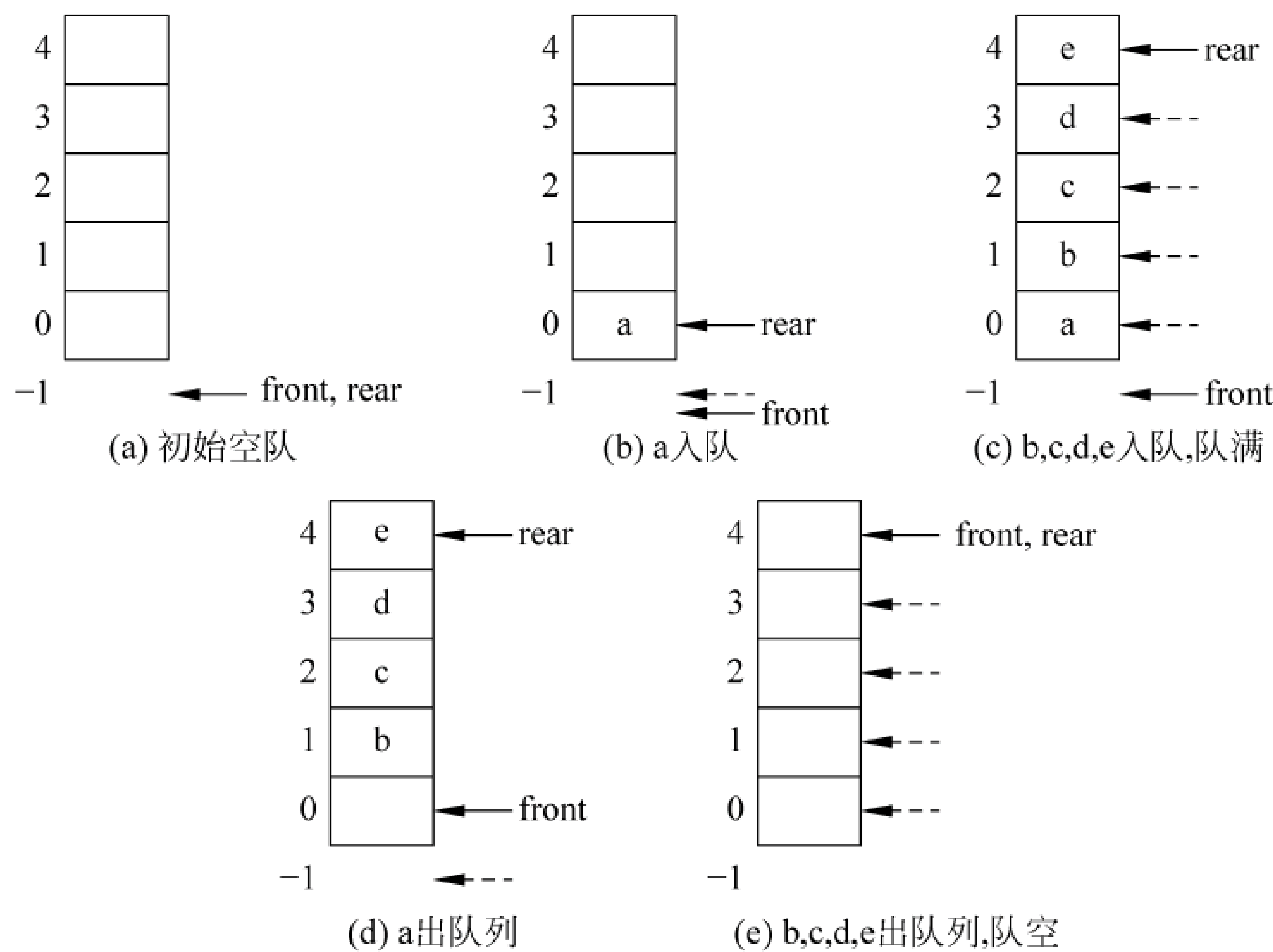


图 3.10 顺序队列操作示意图

1. 顺序队列的基本运算

实现队列的基本运算算法如下。

1）初始化队列

构造一个空队列 Q。将 front 和 rear 指针均设置成初始状态，即-1。

```
void InitQueue(SqQueue * &Q)
{
    Q=(SqQueue * )malloc(sizeof(SqQueue));
    Q→front=Q→rear=-1;
}
```

注意：队列初始空状态和空状态有区别，表述条件不同。

2）销毁队列

释放队列 Q 所占用的存储空间。

```
void DestroyQueue(SqQueue * &Q)
{
    free(Q);
}
```

3）判断队列是否为空

```
bool QueueEmpty(SqQueue * Q)
{
    return(Q->rear==Q->front);
}
```

4）入队列

在队列不满的条件下，先将队尾指针 rear 增 1，然后将元素 e 添加到该位置。

```
bool enQueue(SqQueue * &Q, elemtype e)
{
    if(Q->rear-Q->front==MaxSize)         //或 Q->rear ==MaxSize-1 队满
       return false;                      //返回 false
    Q->rear++;                            //队尾增 1
    Q->data[Q->rear]=e;                   //在 rear 位置插入元素 e
    return true;                          //返回 true
}
```

5）出队列

在队列 Q 不为空的条件下，将队首指针 front 增 1，并将该位置的元素值赋给变量 e。

```
bool deQueue(SqQueue * &Q, elemtype &e)
{
if(Q->rear-Q->front==0)           //队空下溢出
      return false;
Q->front++;                       //队头增 1
e=Q->data[Q->front];              //将 front 位置的元素赋给 e
return true;
}
```

2. 在环形队列中实现队列的基本运算



视频讲解

在前面的顺序队列操作中，元素进队时队尾指针增加 1，元素出队时队头指针增加 1，当进队 MaxSize 个元素后，队满的条件(即 Q->rear-Q->front== MaxSize 或 Q->rear==MaxSize-1)成立，如图 3.10(c)所示。此时即使出队若干元素，队满条件仍然成立(实际上，队列中有空位置)，这是一种**假溢出**，如图 3.10(d)和图 3.10(e)所示。如何解决这种“存在空闲却无法入队列”的假溢出现象？前面使用过的元素入队列操作步骤：先 Q->rear=Q->rear+1，然后元素入队列；当 rear 处于[0,4]范围内，操作均有效，但 Q->rear=Q->rear+1 运算会使得 rear 的取值范围处于[0,+∞)，显然，当 rear 超出 4(即 MaxSize-1)后，就没有操作意义了，因此，通过数学取余的方法使得 rear 被成功地

控制在[0,4](即[0,MaxSize−1])范围内。于是,rear 队尾指针的移动过程更改为 Q→rear=(Q→rear+1)%5,即 Q→rear=(Q→rear+1)%MaxSize。这样就把数组的前端和后端连接起来,形成一个环形的顺序表,即存储队列元素的表从逻辑上看是一个环,称为**环形队列**(也称循环队列)。

通常将顺序队列通过数学取余运算臆造的一个环状空间称为"循环队列",因此,循环队列本质上就是顺序队列,其指针和队列元素间关系保持不变。元素入队列时,队尾指针 rear 的移动方式为 Q→rear=(Q→rear+1)%MaxSize,当队尾指针 Q→rear==MaxSize−1 后,再前进一个位置就自动到 0,环形队列有效地解决了假溢出问题。从顺序队列出队列操作发现,队首指针 front 的移动情况和队尾指针 rear 的移动情况相似。因此,为了有效地控制其在环形队列中操作,front 的移动方式修改为 Q→front=(Q→front+1)%MaxSize,当队首指针 Q→front==MaxSize−1 后,再前进一个位置就自动到 0。将图 3.10 修改为环形队列后,其操作过程如图 3.11 所示。

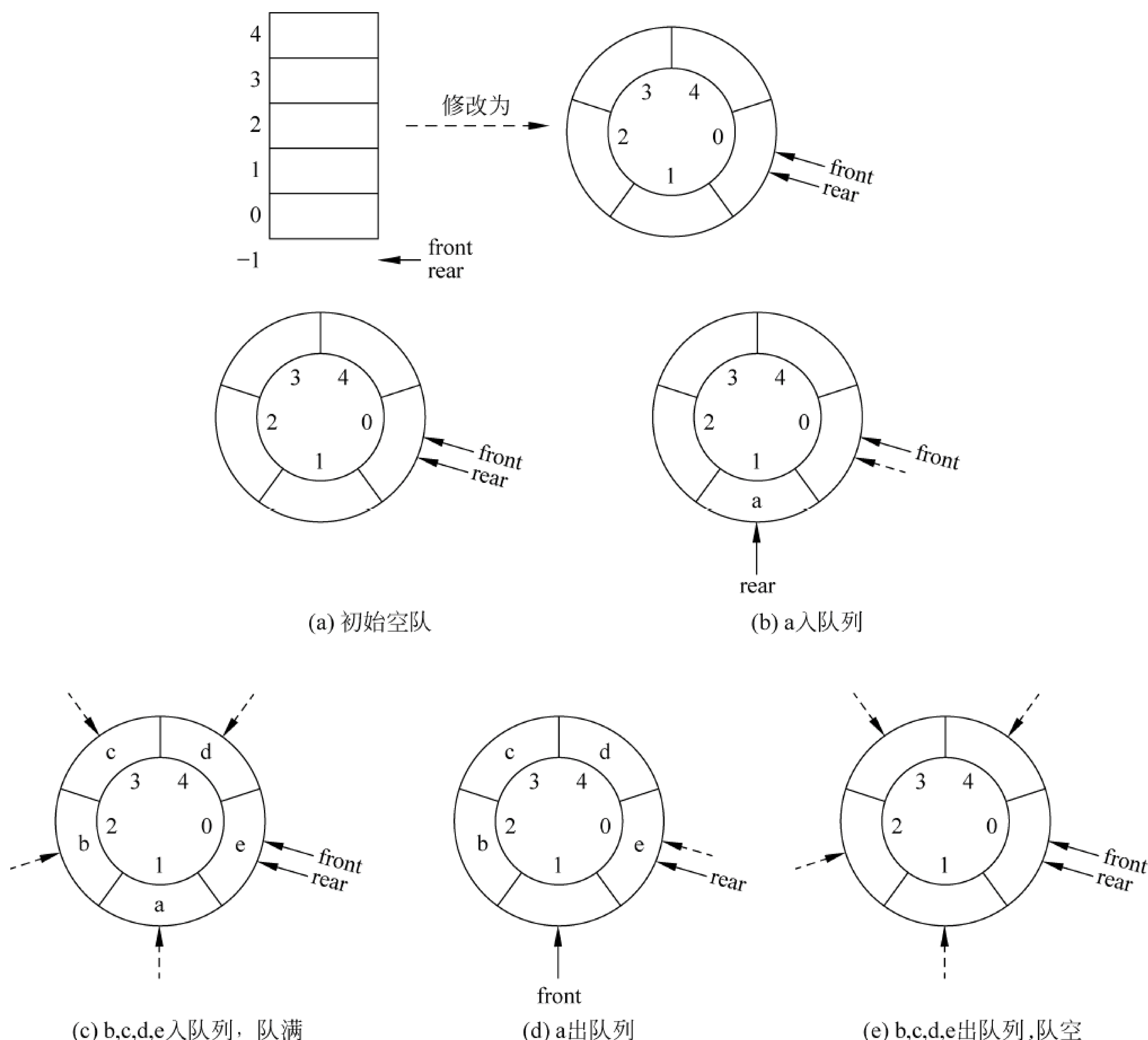


图 3.11 环形队列操作示意图

环形队列的队首指针和队尾指针初始化时都置 0。

```
Q->rear==Q->front=0;
```

进队元素和出队元素时，指针都按逆时针方向进 1。

```
入队列时：Q->rear=(Q->rear+1)%MaxSize;
出队列时：Q->front=(Q->front+1)%MaxSize;
```

那么，环形队列 Q 的队满和队空的判断条件是什么呢？

显然，队空条件是 Q->rear==Q->front。如果入队元素的速度快于出队元素的速度，队尾指针很快就赶上了队首指针，此时可以看出环形队列的队满条件也是 Q->rear==Q->front。

怎样区分这两者呢？通常约定在入队时用一个数据元素空间，队尾指针加 1，等于队首指针时，判断队满，即队满条件为

```
(Q->rear+1)%MaxSize==Q->front
```

队空条件仍为

```
Q->rear==Q->front
```

注意：环形队列能有效解决假溢出问题，但是环形队列分不清队空和队满情况，于是约定 Q->rear==Q->front 为空条件，而牺牲掉一个存储空间作为队满的条件，即(Q->rear+1) % MaxSize==Q->front。

1）初始化队列

构造一个空队列 Q。将 front 和 rear 指针均设置成初始状态，即 0。

```
void InitQueue(SqQueue *&Q)
{
    Q=(SqQueue *)malloc(sizeof(SqQueue));
    Q->front=Q->rear=0;
}
```

2）销毁队列

释放队列 Q 占用的存储空间。

```
void DestroyQueue(SqQueue *&Q)
{
    free(Q);
}
```

3）判断队列是否为空

若队列 Q 满足 Q->rear==Q->front 条件，返回 true；否则返回 false。

```
bool QueueEmpty(SqQueue * Q)
{
    return(Q->rear==Q->front);
}
```

4）进队列

在队列不满的条件下，先将队尾指针 rear 循环增 1，然后将元素添加到该位置。

```
bool EnQueue(SqQueue * &Q, elemtype e)
{
if((Q->rear+1)%MaxSize==Q->front)          //队满上溢出
     return false;
Q->rear=(Q->rear+1)%MaxSize;
Q->data[Q->rear]=e;
return true;
}
```

5）出队列

在队列 Q 不为空的条件下，将队首指针 front 循环增 1，并将该位置的元素赋给 e。

```
bool DeQueue(SqQueue * &Q, elemtype &e)
{   if(Q->rear==Q->front)
         return false;
      Q->front=(Q->front+1)%MaxSize;
      e=Q->data[Q->front];
      return true;
}
```

3.3.3 队列的链式存储

与线性表类似，队列也可以有两种存储表示。用链表表示的队列简称为**链队列**，如图 3.12 所示。一个链队列显然需要两个分别指示队头和队尾的指针(分别称为头指针和尾指针)才能唯一确定。这里，与线性表的单链表一样，为了操作方便，我们也给链队列添加一个头结点，并令头指针指向头结点。

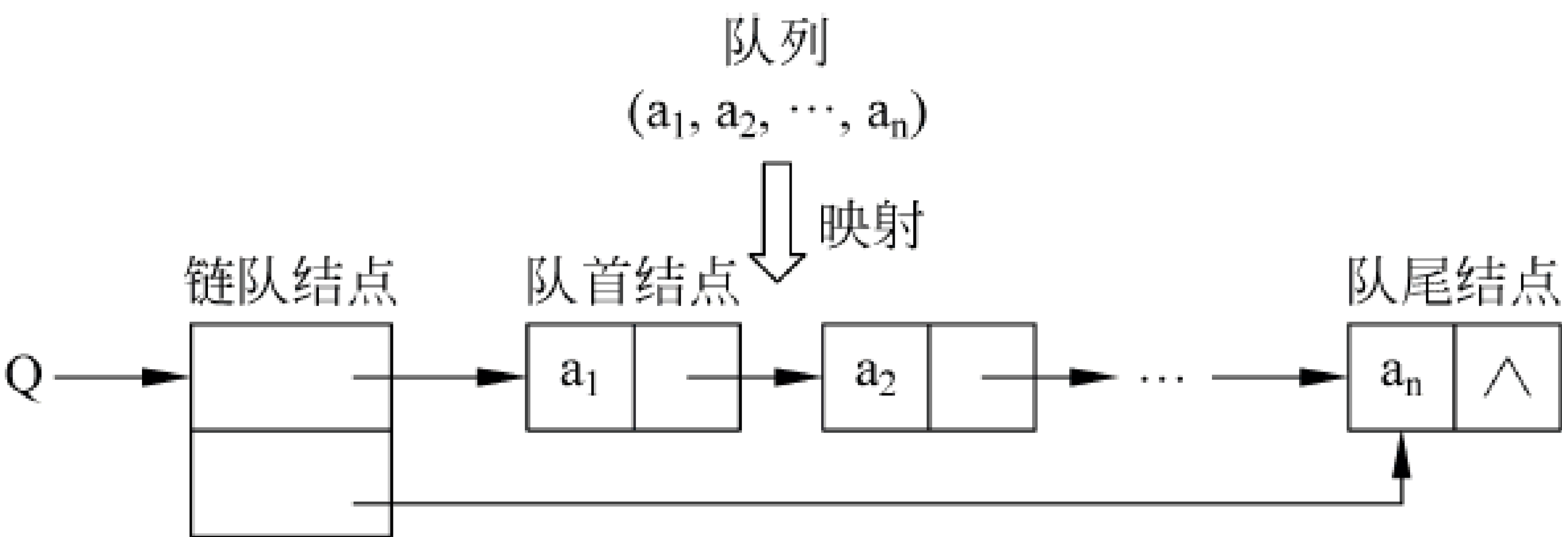


图 3.12　链队列存储结构

链队列中数据结点的类型 QNode 定义如下。

```
typedef struct QNode{
    elemtype data;
    struct QNode * next;
};
```

链队列头结点的类型 LinkQuNode 定义如下。

```
typedef struct
{
    QNode * front;          //队首指针
    QNode * rear;           //队尾指针
}LinkQuNode;
```

图 3.13 所示是一个链队列的动态变化过程。图 3.13(a)是链队列的初始状态；图 3.13(b)是在链队列中插入 3 个元素后的状态；图 3.13(c)是在链队列中删除一个元素后的状态。

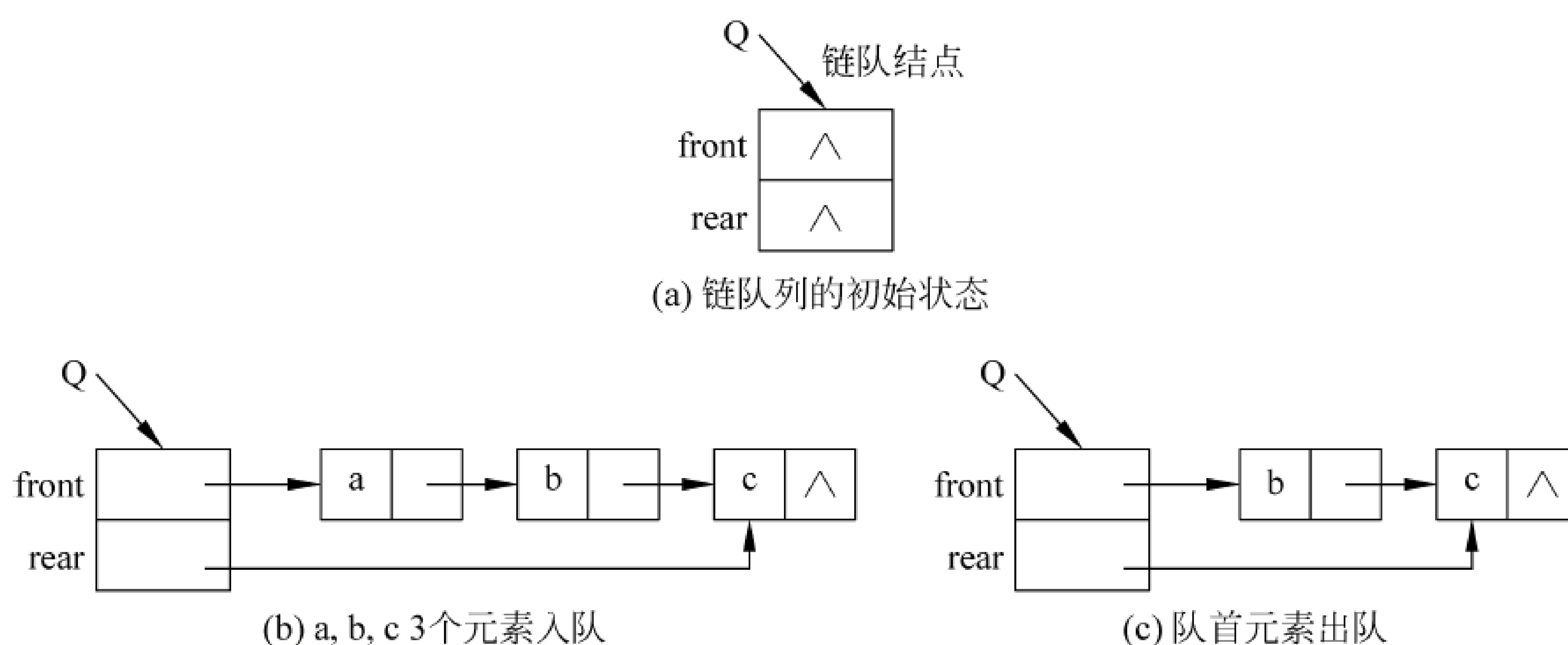


图 3.13　一个链队列的动态变化过程

在以 Q 为链队结点的链队中：队空的条件为 **Q→rear==NULL**；由于只有内存溢出时才会出现队满，而通常不考虑这样的情况，所以看成在链队中不存在的队满；结点 p 入队列的操作是在链队列尾部插入结点 p，并让队尾指针指向它；出队的操作是取出队头所指结点的 data 值并将此结点删除。对应队列的基本运算算法如下。

1）初始化队列

```
void InitQueue(LinkQuNode * &Q)
{
    Q=(LinkQuNode * )malloc(sizeof(LinkQuNode));
    Q->front=Q->rear=NULL;
}
```

2）销毁队列

释放队列占用的存储空间，包括链队列结点和所有数据结点的存储空间。

```
void DestroyQueue(LinkQuNode *&Q)
{
    Qnode *p=Q->front, *r;          //p指向队首结点
    if(p!=NULL)
{
    r=p->next;                      //r指向结点p的后继结点
    while(r!=NULL)                  //r不空循环
    {
    free(p);                        //释放p结点
    p=r;r=r->next;                  //p和r同步后移
    }
    free(p);                        //释放最后一个数据结点
}
    free(Q);                        //释放链队结点
}
```

3）判断链队列Q是否为空

若链队结点的rear域值为NULL，则表示队列为空，返回true；否则返回false。

```
bool QueueEmpty(LinkQuNode *Q)
{
    return(Q->rear==NULL);
}
```

4）进队列

创建data域为e的数据结点p。若原队列为空，则将链接队列结点的两个域均指向p结点，否则将p结点链到单链表的末尾，并让链队结点的rear域指向它。

```
void EnQueue(LinkQuNode *&Q, elemtype e)
{
QNode *p;
p=(QNode *)malloc(sizeof(QNode));
p->data=e;
p->next=NULL;
if(Q->rear==NULL)
    Q->front=Q->rear=p;
else
{ Q->rear->next=p;            //新结点加到队尾
  Q->rear=p;                  //队尾指针指向新结点
 }
}
```

5）出队列

若原队列不为空，则将第一个数据结点的data域值赋给e，并删除该数据结点。若出队之前队列中只有一个结点，则需要将链接队列结点的两个域均设置为NULL，表示队列为空。

```
bool DeQueue(LinkQuNode *&Q, elemtype &e)
{
QNode *t;
if(Q->rear==NULL)              //原来队列为空
    return false;
t=Q->front;                    //t指向首结点
if(Q->front==Q->rear)          //原来队列只有一个数据结点时
    Q->front=Q->rear=NULL;
else                           //原来队列有两个或两个以上数据结点时
    Q->front=Q->front->next;
e=t->data;
 free(t);
return true;
}
```

3.3.4 优先级队列

在3.3.3节介绍的队列中,元素与元素之间不存在优先级次序。也就是说,队列按照严格的FIFO的原则,每次从队列中取出的是最早加入队列的元素。但在实际应用中,队列中的元素可能需要一定的优先级,每次从队列中取出具有特定优先级的元素,这种队列就叫**优先级队列**。在优先级队列中有两个重要操作:插入和删除。插入时只要简单地把一个新元素插入队列中,而删除时则是把最重要的(即优先级最高的)元素从队列中删除。

优先队列的应用比较广泛,如作业系统中的调度程序,当一个作业完成后,需要在所有等待调度的作业中选择一个优先级最高的作业来执行。例如,排队上车,老弱病残者优先上车;排队候诊,危急病人优先就诊;照相馆为顾客洗照片,加钱加急者优先洗等都是优先队列的应用。

每个数据元素的优先级需要根据具体的要求而定。当从优先级队列中删除一个元素时,可能会出现多个优先级相同的元素。在这种情况下,可以把这些优先级相同的元素作为一个一般的先来先服务队列处理。一般设定不出现这种情况。

优先级队列的存储表示和实现方法有很多种。可以采用数组实现,也可以采用链表实现。在每一种表示和实现方法中都使用了一个队列对象存储队列的元素,用参数count标记存放了多少个元素。若采用数组存储优先级队列,则其操作代码如下。

```
#include <iostream>
using namespace std;
const int Size=50;
 typedef struct DataType
{
 int num;
 int priority;                 //优先级
 }datatype;
class P_Queue
{
```

```
private:
datatype data[Size];
int count;                          //计数器
 public:
P_Queue(){ count=0;}
 int empty()
int full()
friend int operator <(datatype &,datatype &);
void InsertPQ(datatype);            //队列的插入
datatype DeQueue();                 //队列的删除
datatype PQueuefront();
int PQueuesize();
void print(datatype x)
{ cout<<x.num<<" "<<x.priority<<endl; }
};
int P_Queue:: empty()
{return count==0; }
int P_Queue:: full()
{return count==Size;}
int operator <(datatype &b ,datatype &c)
{ return b.priority<c.priority;}
void P_Queue::InsertPQ(datatype x)
{
if(full()){cout<<"队列已满!"<<endl;exit(1);}
data[count]=x;
count++;
}
datatype P_Queue::DeQueue()
{
if(empty()){cout<<"队列空!"<<endl;exit(1);}
datatype min=data[0];
int minindex=0;                     //minindex 作为最高优先级的下标
for(int i=0;i<count;i++)
if(data[i]<min)
{ min=data[i];minindex=i;}
data[minindex]=data[count-1]; //把最后一个元素放在要删除元素的位置
count--;
return min;
}
datatype P_Queue:: PQueuefront()
{
if(empty()){cout<<"队列空!"<<endl;exit(1);}
datatype min=data[0];
for(int i=0;i<count;i++)
if(data[i]<min)
{ min=data[i];}
return min;
}
int P_Queue:: PQueuesize(){ return count;}
```

```
void main()
{
P_Queue * p;
p=new P_Queue;
datatype x;
int t=-1;
cout<<"选项: 1.插入 2.删除 3.队列首元素 4.队列大小"<<endl;
while(1)
{
cout<<"输入选项: "; cin>>t;
if(t==1)
{ cout<<"输入元素: ";
cin>>x.num >>x.priority;
p->InsertPQ(x);
}
else if(t==2)
{
p->deQueue();
cout<<"删除成功!"<<endl;
}
else if(t==3)
{ p->print(p->PQueuefront());
}
else if(t==4)
{
cout<<p->PQueuesize()<<endl;
}
else
cout<<"请重新输入:"<<endl;
}
}
```

由于插入操作是直接将元素插入到优先级队列的队尾,因此其运算时间复杂度为 O(1),但用户删除操作需要先扫描整个数组确定最小值元素及其位置,所以其时间复杂度为 O(n),n 是优先级队列的当前元素个数。

3.4 STL 中的栈与队列

3.4.1 STL 中的栈

在 STL 中的 stack 类可作为栈使用。stack 作为一种先进后出的数据结构,它只要一个出口,stack 允许新增元素、移除元素、取得最顶端元素,但除了最顶端以外,没有任何其他方法可以存取 stack 中的元素,即 stack 不允许遍历行为。

STL 中的 stack 类实际上是一种适配器(adapter),它不能被归类为容器(container),而被归类为 container adapter。下面介绍 stack 类的使用。

C++ STL 栈(stack)的头文件为

```
#include <stack>
```

C++ STL 栈(stack)的成员函数介绍如下。

empty():堆栈为空,则返回 true。

pop():移除栈顶元素。

push():在栈顶增加元素。

size():返回栈中的元素数目。

top():返回栈顶元素。

下面给出简单的代码示例,说明上面函数的使用过程。

```
#include "stdafx.h"
#include <stack>
#include <vector>
#include <deque>
#include <iostream>
using namespace std;
int main()
{
 stack<int> mystack;
 //入栈,出栈
 mystack.push(1);
 mystack.push(2);
 mystack.push(3);
 mystack.pop();
 count<<mystack.top()<<endl;
 cout<<mystack.size()<<endl;
 cout<<mystack.empty()<<endl;
 return 0;
}
```

3.4.2 STL 中的队列

同 stack 相同,queue 也是一种容器适配器,是通过简单地修饰 deque 的接口而形成的另一种容器类型。queue 模板类的定义在<queue>头文件中。C++ STL 队列(queue)的头文件为

```
#include <queue>
```

C++ STL 队列(queue)的成员函数介绍如下。

push(x):将 x 压入队列的末端。

pop():弹出队列的第一个元素(队首元素),注意此函数并不返回任何值。

front():返回第一个元素(队首元素)。

back():返回最后被压入的元素(队尾元素)。

empty():当队列为空时,返回 true。

size()：返回队列的长度。

下面给出简单的代码示例,说明上面函数的使用过程。

```
#include <cstdlib>
#include <iostream>
#include <queue>
using namespace std;
int main()
{
int e,n,m;
queue<int> q1;
 for(int i=0;i<10;i++)
     q1.push(i);
 if(!q1.empty())
    cout<<"dui lie bu kong\n";
 n=q1.size();
 cout<<n<<endl;
 m=q1.back();
 cout<<m<<endl;
 for(int j=0;j<n;j++)
 {
    e=q1.front();
    cout<<e<<" ";
    q1.pop();
  }
 cout<<endl;
if(q1.empty())
    cout<<"dui lie bu kong\n";
system("PAUSE");
return 0;
}
```

3.4.3 STL 中的优先队列的使用方法

优先队列(priority_queue)容器与队列一样,只能从队尾插入元素,从队首删除元素。但是,它有一个特性,就是队列中最大的元素总是位于队首,所以,出队时并非按照先进先出的原则进行,而是将当前队列中最大的元素出队。这点类似于给队列里的元素进行了由大到小顺序的排列。元素的比较规则默认按元素值由大到小排序,可以重载“<”操作符重新定义比较规则。C++ STL 优先队列(priority_queue)的头文件为

```
#include <queue>        //queue 与 priority_queue 均被包含在头文件<queue>中
```

C++ STL 优先队列(priority_queue)的成员函数介绍如下。

empty()：如果队列为空,返回 true。

pop()：删除队首元素。

push()：加入一个元素。

size()：返回优先队列中拥有的元素个数。

top()：返回优先队列的队首元素。

在默认的优先队列中，优先级高的先出队。在默认的 int 型中，先出队的为较大的数。下面给出简单的代码示例，说明上面函数的使用过程。

```
#include <cstdlib>
#include <iostream>
#include <queue>
using namespace std;
void main()
{
 priority_queue<int>Q;
 Q.push(1);
 Q.push(5);
 Q.push(2);
 Q.push(3);
 Q.push(6);
 Q.push(4);
 int size=Q.size();
 for(int i=0;i<size;i++)
 {
 cout<<Q.top()<<endl;
 Q.pop();
}
 cout<<endl<<Q.empty()<<endl;
}
```

注意：上述优先级队列编译执行的结果为

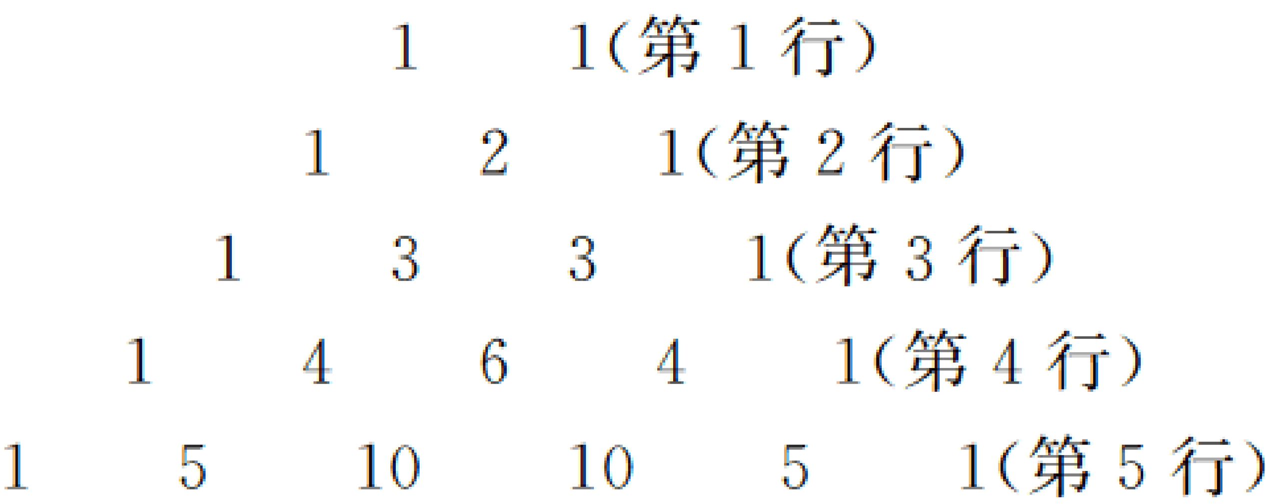
```
6 回车 5 回车 4 回车 3 回车 2 回车 1 回车
```

3.5 队列综合案例

3.5.1 打印杨辉三角形

1. 问题描述

将二项式$(a+b)^n$展开，其系数构成杨辉三角形。杨辉三角形的构造方式是将三角形每一行两边的元素置为 1，其他元素为这个元素“肩”上的两元素之和。例如，一个简单的五行杨辉三角形如下所示。



2. 解题思路

按行将二项式(a+b)^{n}展开式系数的前 n 行打印出来。从三角形的形状可知,除第 1 行以外,在打印第 i 行时,用到上一行(第 i−1 行)的数据,在打印第 i+1 行时,又用到第 i 行的数据。在第 i 行上有 i+1 个数,除了第一个和最后一个数为 1 外,其余的数为上一行中位于该数左、右的两数之和。

若采用循环队列打印输出杨辉三角形前 n 行的值,应设置循环队列的最大空间为 n+2,假设队列中已存有第 i 行的值,为计算方便,在两行之间均加一个“0”作为行间的分隔符,则在计算第 i+1 行前,头指针正好指向第 i 行的“0”,而尾元素为第 i+1 行的“0”。由此从左至右输出第 i 行的值,并将计算所得的第 i+1 行的值插入队列。

如第四行为:0　　1　　4　　6　　4　　1

则第五行为:0　　1　　5　　10　　10　　5　　1

分析第 i 行元素与第 i+1 行元素的关系如图 3.14 所示。

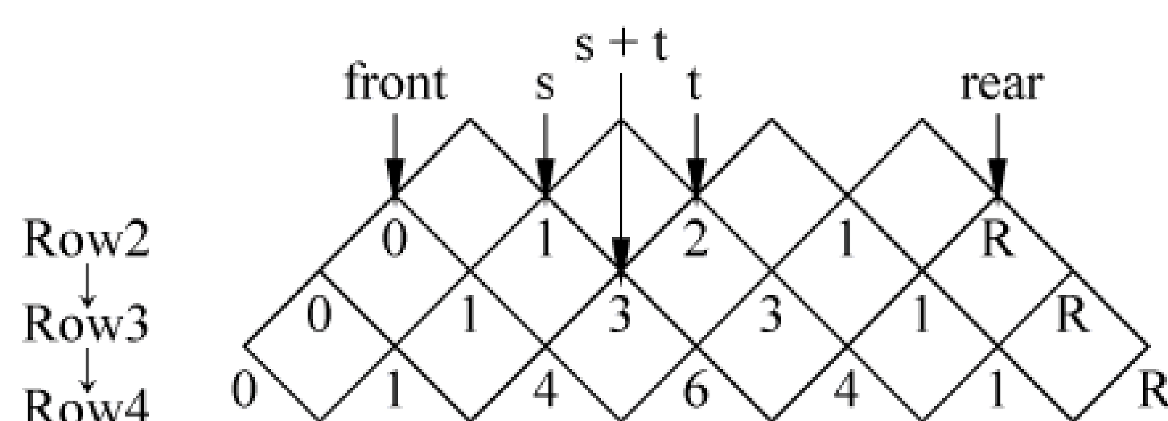


图 3.14　行间关系图

i=2 时,队列的头指针指向 0,尾指针指向 1 的下一位,接下来如何由第二行得到第三行?

首先,第一步:第三行的'0'入队列;第二步:队首元素'0'出队列并送入 s 中;第三步:取队首元素'1'并送入 t 中;第四步:s+t 的值'1'入队列。此时队列的队头指针指向'1'元素,队尾指针指向第三行的第一个 3 的位置。重复第二、三、四步就得到第三行;以此类推,由第三行又得到第四行。按照规律计算出的元素,依次入队列操作,如图 3.15 所示。

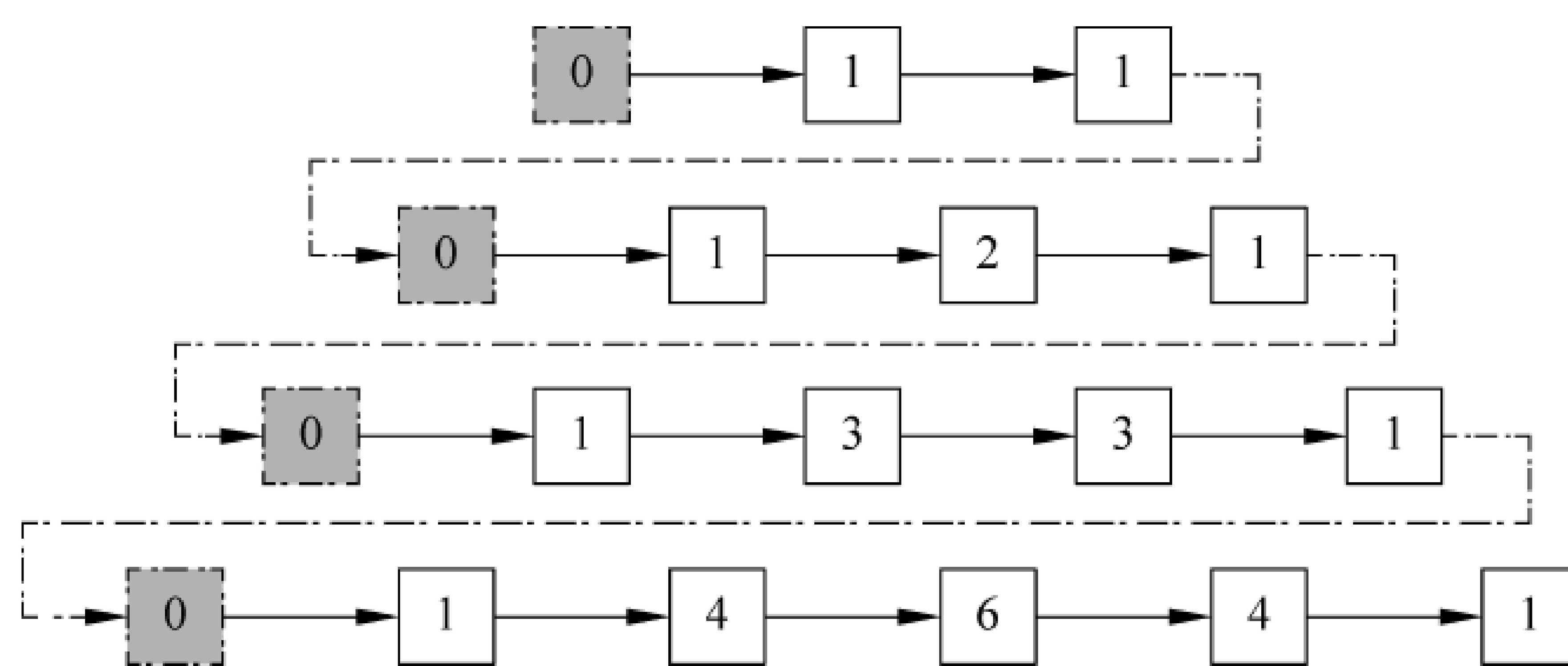


图 3.15　杨辉三角形元素入队顺序

3. 代码实现

```
void YangHui(int n) {
  SqQueue *q;
  int i, j, s, t;
```

```
    printf("1\n");                    //设置杨辉三角形最顶端为 1
    InitQueue(q);                     //设置容量为 n+2 的空队列
    EnQueue(q, 0);                    //设置行分隔符 0,并入队列
    EnQueue(q, 1);
    EnQueue(q, 1);                    //第一行的值入队列
    for(i = 1; i < n; i++) {
      for(j = 1; j < n-i; j++)
        EnQueue(q, 0);                //行分隔符 0 入队列
      do {
        s = DeQueue(q);               //输出队首元素并赋值给 s
        t = GetHead(q);               //取队头元素并赋值给 t
        if(t)
          printf(" % 3d", t);
        else
          printf("\n");
        EnQueue(q, s + t);            //对应的下一行元素 s+t 入队列
      } while(t != 0);
    }
    DeQueue(q);
    printf(" %3d", DeQueue(q));       //输出第 n 行的第一个元素
    while(!QueueEmpty(q)) {           //输出第 n 行的其余元素
      t = DeQueue(q);
      printf(" % 3d", t);
    }
}
```

C++代码实现如下。

```
#include "LinkQueue.h"
using namespace std;

template<class T>
void evaluate(LinkQueue<T> &ori, LinkQueue<T> &target) {
  ori.MakeEmpty();
  while(!target.IsEmpty()) {
    ori.EnQueue(target.DelQueue());
  }
}

int main(int argc, char * * argv) {
  cout << "请输入杨辉三角形的行数 n:";
  int n;
  cin >> n;
  LinkQueue<int> ori;
  ori.EnQueue(1);
  ori.EnQueue(1);
  LinkQueue<int> next;
```

```
        for(int i = 0; i < n-2; i++) {
            next.EnQueue(1);
            while(!ori.IsEmpty()) {
                int i = ori.DeQueue();
                if(!ori. IsEmpty())
                    next.EnQueue(i + ori.GetFront());
                if(ori. IsEmpty())
                    next.EnQueue(i);
            }
            evaluate(ori, next);
        }
        cout << "杨辉三角形" << "行内容如下:" << endl;
        while(!ori.IsEmpty())
            cout << ori.DeQueue() << " ";
        cout << endl;
        return 0;
    }
```

3.5.2 报数问题

1. 问题描述

报数问题即约瑟夫环问题，n 个人围成一圈，给他们随机编号，然后从某个人开始按次序报数，当报到 m 时，这个人出列，从此不断循环，直到圈中只剩最后一个人停止。

2. 解题思路

采用循环队列解决此问题时，从队头开始报数，报数 1 到 m－1 的人先从队列头出队，将出队列的元素再从队尾入队，报数到 m 的人出队列，不再入队列；并从下一个人开始接着从 1 开始报数；继续上述过程，直到队列为空为止。例如，当 n＝10，m＝4 时，依次出列的人分别为 4、8、2、7、3、10、9、1、6、5，则 5 号位置的人为胜利者。

3. 代码实现

```
#include "SqQueue.h"
using namespace std;

int main(int argc, char * * argv) {
    int n, m, i = 1;
    SqQueue Q;
    elemtype e;
    cout << "请输入 n 个人(n<=100):";
    cin >> n;
    if(n > 100 || n < 1) {
        cout << "输入数据错误!";
        return 0;
    }
    InitQueue(Q, n);
```

```
    while(i <= n) {              //入队列
        EnQueue(Q, i);
        i++;
    }
    cout << "\n此时序列顺序为:";
    QueueTravese(Q);
    cout << "\n请输入第m个人出队:";
    cin >> m;
    if(m > n || m < 1) {
        cout << "m输入错误!";
        return 0;
    }
    cout << endl;
    int Count = n;               //Count用来记录剩下的人数
    while(Count != 1) {
        i = 1;
        while(i != m) {
            Q.front = (Q.front + 1) % (Q.MaxSize-1);
            if(Q.base[Q.front] != 0)
                i++;
        }
    }
    DeQueue(Q, e);
    while(Q.base[Q.front] == 0) {
        Q.front = (Q.front + 1) % (Q.MaxSize-1);
        cout << "序号:" << e << "出局!\n";
        cnt--;
    }
    DeQueue(Q, e);
    cout << "\n最后一个是:" << e << endl;
    return 0;
}
```

3.5.3 舞伴问题

1. 问题描述

假设在周末舞会上,男士们和女士们进入舞厅时各自排成一队。跳舞开始时,一次从男队和女队的队头上各出一人配成舞伴。若两队初始人数不相同,则较长的那一队中未配对者等待下一轮舞曲。要求写一算法,模拟上述舞伴配对问题。

2. 解题思路

舞伴问题是用队列进行模拟的典型问题。首先建立两个队列 M 与 F,分别用来存放男、女舞伴。接下来向队列中输入到达舞会的实际人数,当男伴多于女伴时,M 将长于 F,否则 F 将长于 M。当全部的舞伴进入队列后,开始输出配对结果。依次在 M 和 F 的队头分别取出一名男士与一名女士配对,若最后两个队列全部为空,则说明没人剩下,全部配成功;若其中一支队伍为空,而另外一支队伍有剩余,则有剩余的那支队伍中的舞伴在本轮舞会中落单。此时程序输出(或标记)非空队列中第一个人的姓名,表示下一轮舞曲开始时,被第一

个配对的将是此人。

```
#include "LinkQueue.h"
#include <string>
using namespace std;
struct dancer {
    string name;
    char sex;
};

int main(int argc, char * * argv) {
    cout << "请输入总人数:";
    int num;
    cin >> num;

    LinkQueue<dancer> M;
    LinkQueue<dancer> F;
    for(int i = 0; i < num; i++) {
        cout << "请输入人的性别(f or m)及姓名:";
        char sex;
        cin >> sex;
        string name;
        cin >> name;
        dancer newdancer;
        newdancer.name = name;
        newdancer.sex = sex;
        if(sex == 'f')
            F.EnQueue(newdancer);
        if(sex == 'm')
            M.EnQueue(newdancer);
    }
    while(!M.IsEmpty() && !F.IsEmpty())
        cout << M.DelQueue.name << "\t* * *\t" << F.DeQueue().name << endl;
    if(!M.IsEmpty()) {
        cout << "Mr. " << M.GetFront().name << "is waiting !" << endl;
    } else if(!F.IsEmpty()) {
        cout << "Ms. " << F.GetFront().name << "is waiting!" << endl;
    } else
        cout << "ok!" << endl;
    return 0;
}
```

本 章 小 结

本章主要介绍了栈和队列的基本知识,主要学习要点如下。

- 理解栈的定义和相关概念,掌握栈的特点并进行实践。
- 理解顺序栈和顺序表的关联,顺序栈的类型定义。

- 掌握顺序栈的基本操作并将栈结合实践问题进行应用。
- 理解队列的定义和相关概念,掌握队列的特点并进行实践。
- 理解顺序栈和顺序表的关联,顺序栈的类型定义。
- 掌握环形队列产生的原因和基本操作过程。
- 了解链栈和链队列的类型定义和基本操作。
- 掌握栈和队列的基本应用。

第4章 串

计算机中,除了对数值数据进行操作以外,很多时候还要对字符串这类非数值数据进行处理。字符串(简称串)是一种线性结构,在计算机处理非数值问题时占有重要的地位,如信息检索系统、验证用户的输入和创建格式化字符串都会用到字符串。本章主要介绍串的概念、串对应的抽象数据类型、模式匹配算法以及KMP算法。

4.1 串的基本概念和抽象数据类型



视频讲解

4.1.1 串的基本概念

串(String)(或字符串)是指由一对双引号括起来的零个或多个字符组成的有限序列。一般表示为 S="$a_1a_2\cdots a_n$"。不难发现,串的定义与线性表十分相似。事实上,串是一种特殊的线性表,其特殊性体现在组成串的每个数据元素为单个字符,所以线性表的相关知识可以迁移应用到串上。

串的表示中,双引号并不是串本身的内容,它只是起到定界符的作用,用以区分串常量和串变量。例如,a可以看成是一个变量,但"a"则是一个串。其中,S是串的名,双引号括起来的部分称为串的值。串的值可以是英文字母、数字0~9、常用标点符号以及空格等。通常,由ASCII码表示的所有字符均可作为字符串值的组成部分。

双引号中括起来的字符的个数称为串的**长度**。如果串的双引号里没有字符,则称该串为空串,长度为0,符号Ø表示空串。如果串的双引号中的字符由一个或多个空格组成,则称该串为**空格串(又称空白串)**,长度为串中空格的个数。假设某字符存在于串中,则默认该字符在串中第一次出现的位置称为**字符在串中的位置**。例如,字符'i'在串"this is"中的位置是3。

串中任意连续的字符所组成的字符序列称为该串的**子串**。包含子串的串相应地称为**主串**。子串的第一个字符在主串中的位置称为**子串在主串中的位置**。

例如,假设A、B、C、D为如下的4个串。

A="QIN",　B="DAO",

C="QINDAO",　D="QIN DAO"

则它们的长度分别为3,3,6和7;并且A和B都是C和D的子串,A在C和D中的位置都是1;而B在C中的位置是4,在D中的位置则是5。

注意:字符串中第一个字符的位置记为1(位序),而有些教材中则记录为0(下标)。

当且仅当两个串长度相等且对应位置上的字符完全相同时,称两个**串相等**。如:串

"abc"和串"abc"是相等的,但"Abc"和"abc"则是不相等的。上例中的串 A、B、C 和 D 彼此都不相等。

4.1.2 串的抽象数据类型

串的抽象数据类型的定义如下。

ADT String
{
数据对象:D={a_i | 1≤i≤n, n≥0, a_i 为 char 类型}
数据关系:R={<a_{i-1}, a_i> | a_{i-1}, a_i∈D, i=2, …, n}
基本操作:
StrAssign(&T, chars):串赋值。
　　操作结果:将串 chars 赋给串 T,即生成其值等于 chars 的串 T。
StrCopy(&S, chars):串复制。
　初始条件:串 S 存在。
　操作结果:将串 chars 赋给串 s
StrEqual(S, T):判断串是否相等。
　初始条件:串 S 和串 T 已知存在。
　操作结果:若串 S 和串 T 相等,则返回 true,否则返回 false。
StrLength(S):求串的长度。
　初始条件:串 S 已知存在。
　操作结果:统计并返回串 S 里面字符的个数。
Concat(S1, S2):串连接。
　初始条件:串 S1 和串 S1 已知存在。
　操作结果:返回由串 S1 和串 S2 连接在一起形成的新的字符串。
SubStr(S, pos, len):求子串。
　初始条件:串 S 已知存在。
　操作结果:返回串 S 从 pos(1≤pos≤n)位置开始的 len 个字符形成的新串。
InsStr(S, pos, T):子串插入。
　初始条件:串 S 和串 T 已知存在。
　操作结果:将串 T 插入串 S 的第 pos(1≤pos≤n+1)个位置,并返回新串。
DelStr(S, pos, len):子串删除。
　初始条件:串 S 已知存在。
　操作结果:从串 S 中的第 pos(1≤pos≤n)个位置开始的 len 个字符。
RepStr(S, pos, len, T):子串替换。
　初始条件:串 S 和串 T 已知存在。
　操作结果:串 S 的第 pos(1≤pos≤n)个位置开始的 len 个字符由串 T 替换,并返回新串。
DispStr(S):串输出。
　初始条件:串 S 已知存在。
　操作结果:输出串 S 的所有字符值。
DestroyStr(&S):串销毁。
　初始条件:串 S 已知存在。
　操作结果:释放为串 S 分配的存储空间。
Index(S, T):串的模式匹配。
　初始条件:串 S 和串 T 已知存在。
　操作结果:S 为目标串,T 为模式串。若从 S 串中能找到子串与 T 串相等,则匹配成功,返回位置;否则匹配失败,并返回-1。
　} ADT String

4.2 串的存储结构

串作为特殊的线性表,故其存储结构与线性表的存储结构类似,也分为顺序存储和链式存储。顺序存储结构的串称为顺序串。链式存储结构的串称为链串。

4.2.1 串的顺序存储结构——顺序串

顺序串中的字符被依次存放在一组连续的存储单元里。通常,一个字节(Byte),即8位(bit)可以表示一个字符(存放该字符的ASCII码)。而计算机内存是按字进行寻址的,即以字为存储单元,一个存储单元指的是一个字。而一个字可能包含多个字节,具体字节数因机器而异,这样,一个存储单元就可以存放多个字符。

顺序串的存储方式通常有两种:**非紧缩格式和紧缩格式**。非紧缩格式是指每个字只存放一个字符,如图4.1所示(假设当前字长为3个字节)。紧缩格式则是指每个字存放多个字符,如图4.2所示。

1001	a		
1002	b		
1003	c		
1004	d		
1005	e		
1006	f		
1007	g		
1008	h		
1009	i		
100a	j		

图4.1 非紧缩式存储

1001	a	b	c
1002	d	e	f
1003	g	h	i
1004	j	#	#
1005			
1006			
1007			
1008			
1009			
100a			

图4.2 紧缩式存储

例如:字符串S="abcdefghij"分别采用非紧缩格式和紧缩格式的顺序存储,如图4.1和图4.2所示。

串的紧缩格式的特点是:存储密度大,节省存储空间,但算法操作较复杂,处理单个字符不方便,运算效率低,因为需要花费时间从同一个字中分离字符;相反,串的非紧缩格式的特点是:存储密度小,比较浪费存储空间,但算法操作简单,处理单个字符或者一组连续字符方便。因此,在后续的算法实现中,除特殊声明外,均采用非紧缩式存储方式。

对于非紧缩格式的顺序串,其类型声明如下。

```
typedef struct
{char data[MaxSize];                    //存放串字符
 int length;                            //存放串长
}SqString;                              //顺序串类型
```

下面讨论在顺序串上实现串基本运算的算法。

1. 串赋值

将串 chars 赋给串 T，即生成值等于 chars 的串 T。

```
void StrAssign(SqString &T,char chars[])
{int i;
for(i=0;chars[i]!='\0';i++)
   T.data[i]=chars[i];
T.length=i;
}
```

2. 串复制

将串 T 赋给串 S。

```
void StrCopy(SqString &S,SqString T)
{int i;
for(i=0;i<T.length;i++)
    S.data[i]=T.data[i];
S.length=T.length;
}
```

3. 判断串相等

判断串是否相等，若串 S 和串 T 相等，则返回 true(1)，否则返回 false(0)。

```
bool StrEqual(SqString S,SqString T)
{
bool issame=true;
int i;
if(S.length!=T.length)                    //长度不相等时，返回 0
    issame=false;
else
    for(i=0;i<S.length;i++)
     if(S.data[i]!=T.data[i])             //有一个对应的字符不相同时，返回 0
      { issame=false;
        break;
      }
return issame;
}
```

4. 计算串长度

求串 S 的长度，即双引号里字符的个数。

```
int  StrLength(SqString S)
{   return S.length;   }
```

5. 串连接

串连接，将 S2 连接在串 S1 之后，并返回由串 S1 和串 S2 连接在一起形成的新的字符串。

```
SqString  Concat(SqString S1,SqString S2)
{
SqString str;
int i;
str.length=S1.length+S2.length;
for(i=0;i<S1.length;i++)                    //将 S1.data[0…S1.length-1]复制到 str
    str.data[i]=S1.data[i];
for(i=0;i<T.length;i++)                     //将 S2.data[0…S2.length-1]复制到 str
    str.data[S1.length+i]=S2.data[i];
return str;
}
```

6. 从串中取子串

求子串,返回串 S 从 pos(1≤pos≤n)位置开始的 len 个字符形成的子串。当参数不正确时,返回一个空串。

```
SqString  SubStr(SqString S,int pos,int len)
{
SqString str;
int k;
str.length=0;
if(pos<=0||pos>S.length||len<0||pos+len-1>S.length)
     return str;                           //参数不正确时,返回空串
for(k=pos-1;k<pos+len-1;k++)               //将 S.data[pos…pos+len]复制到 str
     str.data[k-pos+1]=S.data[k];
str.length=len;
return str;
}
```

7. 串插入

子串插入,将串 T 插入串 S 的第 pos(1≤pos≤n+1)个位置,即将 T 的第一个字符作为 S 的第 pos 个字符,并返回产生的新串。参数不正确时,返回一个空串。

```
SqString  InsStr(SqString S,int pos,SqString T)
{
int i;
SqString str;
str.length=0;
if(pos<=0||pos>S.length+1)                 //参数不正确时,返回空串
     return str;
for(i=0;i<pos-1;i++)                       //将 S.data[0…pos-2]复制到 str
    str.data[i]=S.data[i];
for(i=0;i<T.length;i++)                    //将 T.data[0…T.length-1]复制到 str
    str.data[i+pos-1]=T.data[i];
for(i=pos-1;i<S.length;i++)                //将 S.data[pos-1…S.length-1]复制到 str
    str.data[T.length+i]=S.data[i];
```

```
str.length=S.length+T.length;
return str;
}
```

8．串删除

删除从串 S 中的第 pos(1≤pos≤n)位置开始的 len 个字符，并返回产生的新串。当参数不正确时，返回一个空串。

```
SqString  DelStr(SqString S,int pos,int len)
{
int i;
SqString str;
str.length=0;
if(pos<=0||pos>S.length||pos+len>S.length+1)     //参数不正确时，返回空串
    return str;
for(i=0;i<pos-1;i++)                             //将 S.data[0…pos-2]复制到 str
   str.data[i]=S.data[i];
for(i=pos+len-1;i<S.length;i++)                  //将 S.data[pos+len-1…S.length-1]
                                                 //复制到 str
   str.data[i-len]=S.data[i];
str.length=S.length-len;
return str;
}
```

9．串替换

在串 S 的第 pos(1≤pos≤n)个位置开始的 len 个字符由串 T 替换，并返回新串。如参数不正确，则返回一个空串。

```
SqString  RepStr(SqString S,int pos,int len,SqString T)
{
int i;
SqString str;
str.length=0;
if(pos<=0||pos>S.length+1)              //参数不正确时，返回空串
   return str;
for(i=0;i<pos-1;i++)                    //将 S.data[0…pos-2]复制到 str
   str.data[i]=S.data[i];
for(i=0;i<T.length;i++)                 //将 T.data[0…T.length-1]复制到 str
   str.data[i+pos-1]=T.data[i];
for(i=pos-1;i<S.length;i++)             //将 S.data[pos-1…S.length-1]复制到 str
   str.data[T.length+i]=S.data[i];
str.length=S.length+T.length;
return str;
}
```

10．输出串

输出串 S 的所有字符。

```
void  DispStr(SqString S)
{
int i;
if(S.length>0)
{for(i=0;i<S.length;i++)
     printf("%c",S.data[i]);
printf("\n");
 }
}
```

11. 释放串

释放为串 S 分配的存储空间。

```
void  DestroyStr(SqString S)
{
free(&S);
}
```

【例 4.1】 假设串采用顺序存储结构进行存储,请设计一个算法 Strcmp(S,T),按字典顺序比较两个串 S 和 T 的大小。

算法分析如下。

step1:对于串 S 和串 T 在共同长度范围内的对应字符依次比较:

- S 的字符大于 T 的字符时,返回 1;
- S 的字符小于 T 的字符时,返回 −1;
- S 的字符等于 T 的字符时,则按上述规则继续比较。

step2:当串 S 和串 T 在共同长度范围内的字符完全相同时,比较串 S 和 T 的长度:

- 两者长度相等时,返回 0;
- S 的长度大于 T 的长度时,返回 1;
- S 的长度小于 T 的长度时,返回 −1。

对应的算法如下。

```
int  Strcmp(SqString S,SqString T)
{int i,len;
if(S.length<=T.length)                    //if…else 结构用来求 S 和 T 的共同长度
  len=S.length;
else
  len=T.length;
for(i=0;i<len;i++)                        //在共同长度范围内逐个字符进行比较
{ if(S.data[i]>T.data[i])
     return 1;
else if(S.data[i]<T.data[i])
     return -1;
 }
```

```
    if(S.length==T.length)                    //S==T 情况
        return 0;
    else if(S.length>T.length)                //S>T 情况
        return 1;
    else
        return -1;                            //S<T 情况
}
```

【例 4.2】 假设串采用顺序存储结构进行存储。请设计一个算法，求串 S 中出现的第一个最长的由连续相同字符构成的平台。

分析：用 pos 保存在 S 中最长的平台的开始位置，max 保存其长度，先将它们初始化为 0。扫描串 S，计算局部重复子串 length，若比 max 大，则更新 max，并用 pos 记下其开始位置。对应的算法如下。

```
void  LongestString(SqString S,int &pos,int &max)
{
int length=1,i=0,start=0;                 //length 保存平台的长度
pos=0,max=0;                              //pos 保存最大平台在 s 中的开始位置,max 保存其长度
while(i<S.length-1)
 if(S.data[i]==S.data[i+1])
  {i++;
   length++;
   }
else
  { if(max<length)
    { max=length;
     pos=start;}
   i++;start=i;                           //初始化下一个平台的起始位置和长度
   length=1;
 }
}
```

4.2.2 串的链式存储结构——链串

链串的存储形式与一般的链表类似，其主要区别在于，链串中的每一个结点可以存储多个字符。通常，将链串中每个结点中存储的字符的个数称为**结点大小**。图 4.3 和图 4.4 分别表示同一个串“ABCDEFGHIJKLMN”的结点大小为 4(存储密度大)和 1(存储密度小)的链式存储结构。

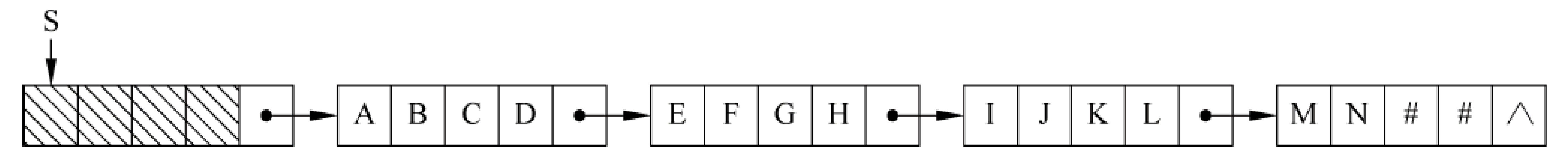


图 4.3 结点大小为 4 的链串

由于串长不一定是结点大小的整数倍，所以当结点大于 1 时，链表中的最后一个结点不一定全被串值占满。如图 4.3 中的最后一个结点所示，此时应在这些未占用的数据域上补

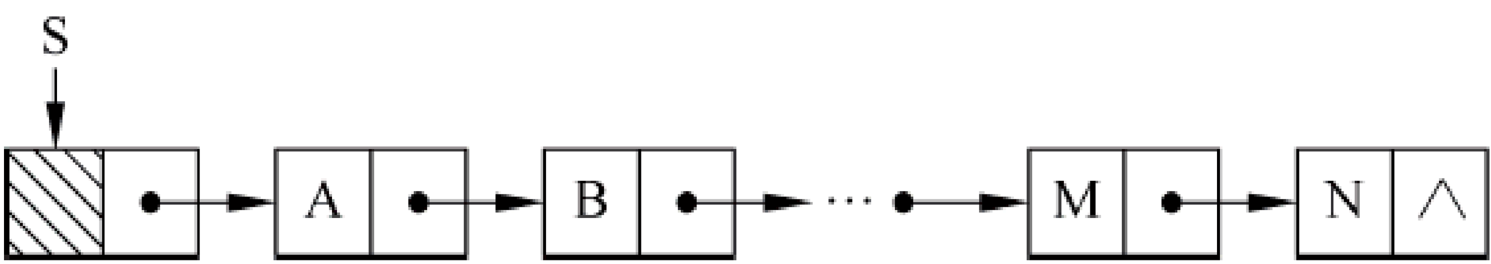


图 4.4　结点大小为 1 的链串

上特殊符号#,以示区别。

链串结点大小的选择与顺序串的格式选择一样重要,它直接影响串处理的效率。在各种串的处理系统中,所处理的串往往很长或很多。例如,一本书的几百万个字符、情报资料的成千上万个条目等,因此,这要求考虑串值的存储密度。链串中,结点越大,存储密度越大,在进行结点插入、删除和替换等操作时需要移动大量的字符,操作不方便。因此,大结点适合采用串的静态使用方式。而结点越小,运算处理越方便,但相应的存储密度会下降。本教材默认规定的链串结点的大小均为 1。存储密度的定义为

存储密度=(串值所占的存储单元)/(实际分配的存储单元)

链串的结点类型定义如下。

```
#define  snodesize    50              //定义结点大小
typedef struct snode                  //定义结点类型 snode
{
char data[snodesize];
struct snode  * next;
}snode;
typedef struct
{
snode   * head;                       //串的头指针
int   len;                            //串实际的长度
}LiString;
```

下面讨论在链串上实现串基本操作的算法。

1. 串初始化

将一个字符串常量 str 赋给串 S,并生成一个值等于 str 的串 S。以下采用尾插法建立链串 S。

```
void  StrAssign(LiString   * &S, char str[ ])
{
int i;
snode  * r, * p;
S=(LiString * ) malloc(sizeof(LiString));
r=S;                                   //r 始终指向尾结点
for(i=0;str[i]!='\0';i++)
{   p=( snode  * ) malloc(sizeof(snode));
    p->data=str[i];
    r->next=p;r=p;
```

```
}
r->next=NULL;
}
```

算法的时间复杂度为 O(n),n 是链串中的结点个数。

2. 串复制

将串 T 复制给串 S。参照链表中的尾插法建立复制后的链串 S。

```
void  StrCopy(LiString  *&S, LiString *T)
{
snode *p=T->next, *q, *r;
S=(LiString *)malloc(sizeof(LiString));
r=S;
while(p!=NULL)
{
  q=(snode *)malloc(sizeof(snode));
  q->data=p->data;
  r->next=q;
  r=q;
  p=p->next;
 }
r->next=NULL;
}
```

算法的时间复杂度为 O(n),n 是链串中的结点个数。

3. 判断串相等

判断两个串是否相等,若两个串 S 与 T 相等,则返回 true,否则返回 false。

```
bool  StrEqual(LiString *S, LiString *T)
{
snode *p=S->next, *q=T->next;
while(p!=NULL&&q!=NULL&&p->data==q->data)
{
  p=p->next;
  q=q->next;
  }
if(p==NULL&&q==NULL)
{  return true;  }
else
   return false;
}
```

算法的时间复杂度为 O(n),n 是链串中的结点个数。

4.3　串的模式匹配

视频讲解

4.3.1　串的古典匹配算法

串的古典匹配算法又称串的简单模式匹配算法,是数据结构中字符串的一种基本运算。给定一个子串,要求在某个字符串中找出与该子串相同的所有子串,这就是模式匹配。

假设 T 是给定的子串,S 是待查找的字符串,其中 T 称为**模式串**,S 称为**目标串**。要求从 S 中找出与 T 相同的所有子串,这个问题称为**模式匹配**问题。如果 S 中存在一个或多个与模式串 T 相等的子串,则算法结果为模式串 T 在目标串 S 中第一次出现的位置,即匹配成功;否则匹配失败,并返回−1。

简单模式匹配算法的思路是**穷举法**,是将 S 串中所有可能的子串逐一和 T 串进行列举验证,直至全部情况验证完成。显然,简单模式匹配的工作重心是放在目标串 S,若串 S 的长度足够大,且有很多不同的子串,当把子串逐个列举出来和模式串 T 进行比对,这个过程显然是枯燥的,但贵在算法实现容易让人理解。若目标串 S="ababcabaabcca",模式串 T="abaa",则简单模式匹配的过程如图 4.5 所示。

```
目标串 S="ababcabaabcca"
模式串 T= "abaa"

简单匹配过程如下：
第1趟: a b a b c a b a a b c c a
       a b a a
第2趟: a b a b c a b a a b c c a
         a b a a
第3趟: a b a b c a b a a b c c a
           a b a a
第4趟: a b a b c a b a a b c c a
             a b a a
第5趟: a b a b c a b a a b c c a
               a b a a
第6趟: a b a b c a b a a b c c a
                 a b a a
```

图 4.5　简单模式匹配的过程

匹配过程中,首先从目标串的第一个字符开始和模式串的第一个字符进行比较,若相等,则继续比较后续字符,否则本趟匹配失败,开始新一趟匹配,于是从目标串的第二个字符开始和模式串的第一个字符进行比较,重复上述步骤,直到匹配成功或者本趟匹配失败,开始新一趟匹配。

因此,设置两个下标 i,j 分别指向 S 串中的字符和 T 串中的字符,若对应位元素 $s_i==t_j$,则继续比较后续字符,否则本趟匹配失败,要进行新一趟的匹配,这时下标 i,j 需要重新定位,找到各自的起始位置;不难发现,下标 j 每趟都回到 T 串开始位置(即 j=0 位置),而 i 每趟回到的位置总比上一趟后移一位。经挖掘发现,某一趟匹配失败时,即 $s_i!=t_j$ 时通过本趟失败的经验"$t_0t_1\cdots t_{j-1}$"="$s_{i-j}s_{i-j+1}\cdots s_{i-1}$"得出下一趟匹配的起始位置 **i=i−j+1**,

j=0。因此，简单匹配算法的整体特点共有两个：一是每趟匹配失败时，模式串在目标串中整体前进一位（通过 i 的起始位置可以发现）；二是每趟匹配失败时，下标 i 和 j 都有回溯现象（j 每趟都要回退到 0 位置，而 i 要回退到前面新的起始位置）。于是，若目标串 S 的长度为 n，模式串 T 的长度为 m（n>>m），则最多匹配（**n-m+1**）趟，每一趟最多比较字符的次数为 m 次，算法最坏情况下时间复杂度为 O(n×m)。简单模式匹配算法分析过程如图 4.6 所示。

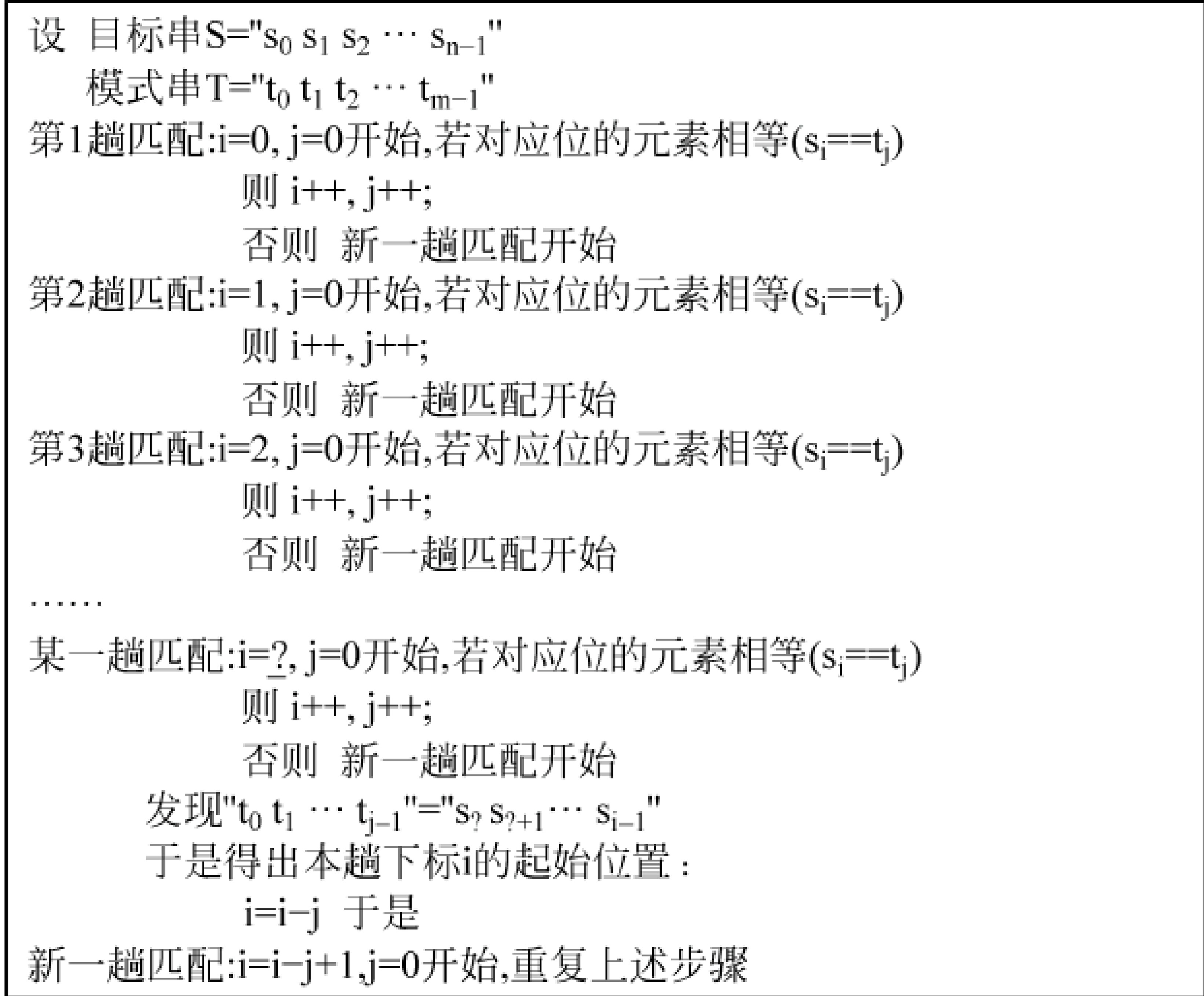


图 4.6 简单模式匹配算法分析过程

算法实现过程如下。

```
int  Index(SqString S,SqString T)
{ int i=0,j=0;                         //定义 i,j 分别作为 S 和 T 的下标
while(i<S.length&&j<T.length)
  {
   if(S.data[i]==T.data[j])            //若本次字符比较相等，则下标 i,j 均加 1
     { i++;j++; }
   else
     { i=i-j+1;j=0; }                  //否则本趟匹配失败，进行下一趟比较
   }
if(j==T.length)                        //匹配成功，返回 T 在 S 中首次出现的位置
     return  i-T.length+1;
else
     return -1;                        //匹配失败，返回-1
}
```

关于简答匹配算法的实现过程，还可以有下面的实现方法。

```
int  Index(SqString S,SqString T)
{
```

```
int i=0,j=0;                          //定义 i,j 分别作为 S 和 T 的下标
 int k=1;                             //定义 k,表示匹配的总次数
 while(i<S.length&&j<T.length)
{
if(S.data[i]==T.data[j])              //若本次字符比较相等,则下标 i,j 均加 1
    {  i++;j++;  }
else
    { i=k;j=0;k++; }                  //否则本趟匹配失败,进行下一趟比较
}
 if(j==T.length)                      //匹配成功,返回 T 在 S 中首次出现的位置
      return k;
 else
      return -1;                      //匹配失败,返回-1
}
```

上面两种算法的功能实现是一致的,前一个算法着重强调匹配的趟数 k 对 i 匹配起始位置的影响,当第 k-1 趟匹配失败时,下标 i=k-1,j=0,重新开始第 k 趟匹配;而后一个算法较常见,不考虑具体是哪一趟匹配,只注重挖掘当前某一趟排序过程对下一趟新的匹配有何影响,前面已经详细说明过,在此不再赘述。例如:目标串 S="ababcabaabcca",模式串 T="abaa",匹配成功且返回结果 6,其实现过程如图 4.7 所示。

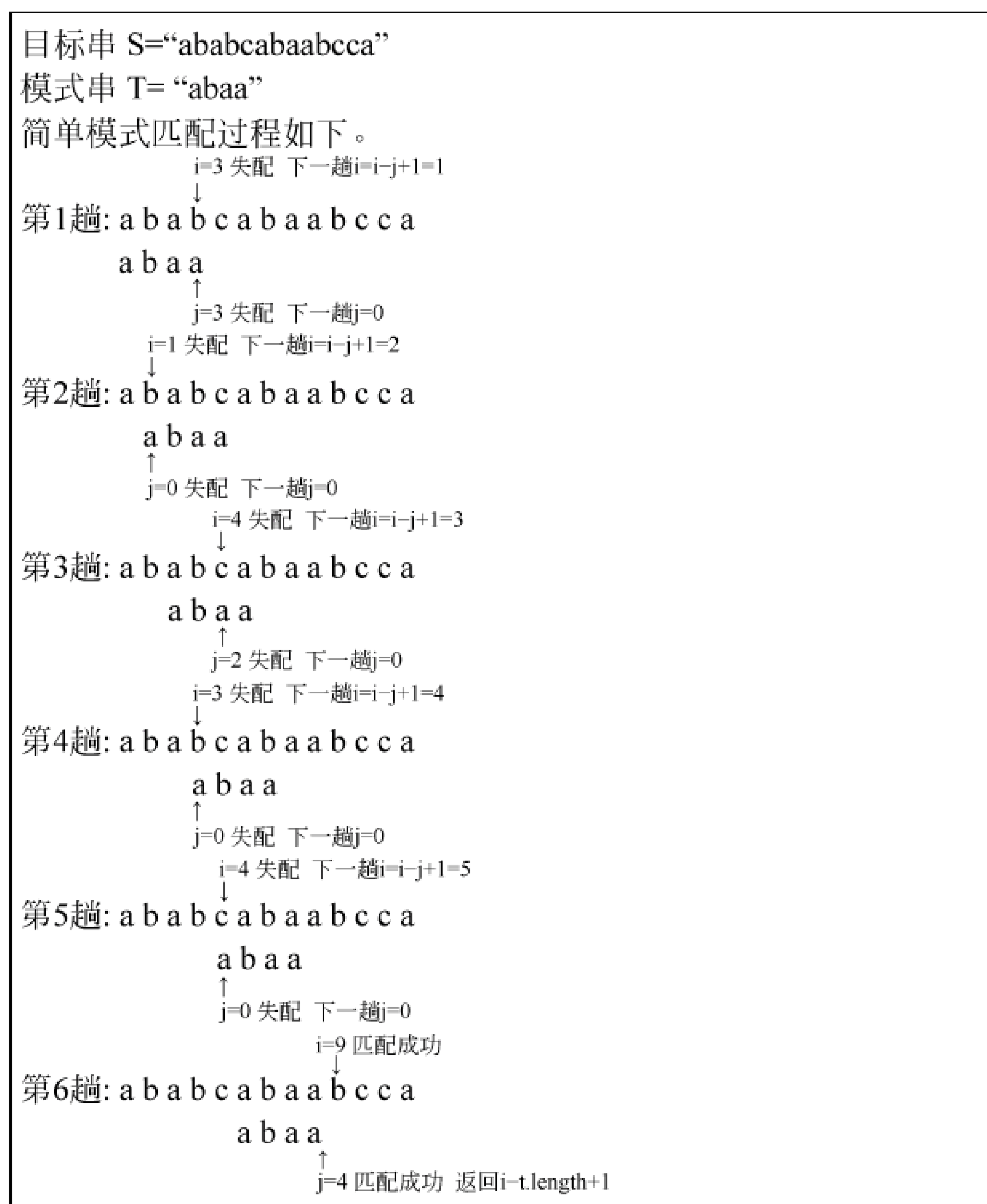


图 4.7 简单模式匹配算法实现过程

显然，简单匹配算法效率较低，若要改进算法，需要从它的两个特点入手进行改进，于是后来人们提出了 KMP 算法。

4.3.2 串的 KMP 算法


视频讲解

KMP 算法是由 D. E. Knuth 与 V. R. Pratt、J. H. Morris 发明的一种快速匹配算法，是对简单匹配算法的一种改进算法。它的特点有两个：一是每趟匹配失败时，模式串 T 在目标串 S 中尽可能多地向后移动(≥1 位)；二是匹配失败时，下标 i 不变(避免回溯)，下标 j 回溯到大于等于 0 的位置。因此，该算法是以 O(n+m)的时间复杂度完成串的模式匹配。KMP 算法的实现过程如图 4.8 所示。

设目标串

　　$S="s_0s_1s_2\cdots s_{n-1}"$

模式串

　　$T="t_0t_1t_2\cdots t_{m-1}"$

某一趟匹配过程中，$t_j!=s_i$

本趟匹配失败，但是$"t_0t_1\cdots t_{j-1}"="s_{i-j}\ s_{i-j+1}\cdots s_{i-1}"$成立

对$"t_0t_1\cdots t_{j-1}"$取真子串运算并再次分析

(1)若$"t_0t_1\cdots t_{j-2}"!="t_1t_2\cdots t_{j-1}"$

　则$"t_0t_1\cdots t_{j-2}"!="s_{i-j+1}s_{i-j+2}\cdots s_{i-1}"$

于是下一趟匹配必失败

(2)若$"t_0t_1\cdots t_{j-3}"!="t_2t_3\cdots t_{j-1}"$

　则$"t_0t_1\cdots t_{j-2}"!="s_{i-j+2}s_{i-j+3}\cdots s_{i-1}"$

于是下下一趟匹配也失败

……

若存在整数k，使得

　　$"t_0t_1\cdots t_{k-1}"="t_{j-k}\ t_{j-k+1}\cdots t_{j-1}"$

　则$"t_0t_1\cdots t_{k-1}"="s_{i-k}\ s_{i-k+1}\cdots s_{i-1}"$

于是该趟匹配有希望成功，开始新一趟匹配

　则 j=k, i不变，重新开始匹配

图 4.8　KMP 算法的实现过程

通过图 4.8 发现，KMP 算法研究的重点在模式串 T 上，和目标串 S 关联不大，通过自身的挖掘找到最大整数 k，从而确定新一趟匹配时 j 的起始位置(即 j=k 位置开始)，于是字符 t_k 和 s_i 开始进行比对，并依次进行字符的比对。此时模式串 T 在目标串 S 中整体前进 j−k位，而 $0\leqslant k\leqslant j-1$。

如何求 k 的值呢？分析表明，$"t_0t_1\cdots t_{j-1}"$前缀串中取出的前 k 个字符和后 k 个字符组成的真子串相等时，才开始进行新一趟匹配，这样计算出的 k 值为最大数值，且 k 的计算只与模式串有关。而模式串 T 是由若干个字符组成的有限序列，在进行匹配时每个字符都有可能出现“失配”情况，因此在不同的位置上失配就会得到各自的 k 值，于是若干 k 值便于保存使用，就被定义成 next[ ]数组。因此，KMP 算法的重点是对模式串 T 进行 next[ ]数组的计算。

若令 next[j]=k，则 next[j]表明当模式串中下标为 j 的字符与目标串中相应下标为 i 字符“失配”时，在模式串中重新定位下标 j 的位置到 k 处，并和目标串中当前位置上的字符重新进行比较。由此可引出模式串的 next 函数的定义。

$$next[j]=\begin{cases}-1 & 当\ j=0\ 时\\ \max\{k \mid 0\leqslant k\leqslant j-1\ 且"t_0t_1\cdots t_{k-1}"="t_{j-k}t_{j-k+1}\cdots t_{j-1}"\} & 当\ j>0\ 时\\ 0 & 其余情况\end{cases}$$

上式中,当 j=0 失配时,next[0]=-1,表示模式串的第一个字符和目标串比对失败,于是不存在所谓的"$t_0t_1\cdots t_{j-1}$"匹配成功经历,此刻让 i 前进一位,j=0 保持不变,进行新一趟匹配;也可以认为此时继续沿用简单匹配方法(即 i=i-j+1=i+1,j=0)进行新一趟匹配。其余情况,next[j]=0,表示在"$t_0t_1\cdots t_{j-1}$"中取出前 k 个字符组成的真子串等于后 k 个字符组成的真子串,长度为 0,前 0 个字符组成的真子串等于后 0 个字符组成的真子串,即 ∅=∅。

由此定义可推出下列模式串"abaa"的 next 函数值。

j	0 1 2 3
模式串	a b a a
next[j]	-1 0 0 1

于是,根据模式串 T="abaa"计算的 next[ ]数组和目标串 S="ababcabaabcca"进行快速匹配的过程如图 4.9 所示。模式串 T 共匹配 4 趟就匹配成功,并返回模式串在目标串中的位置。

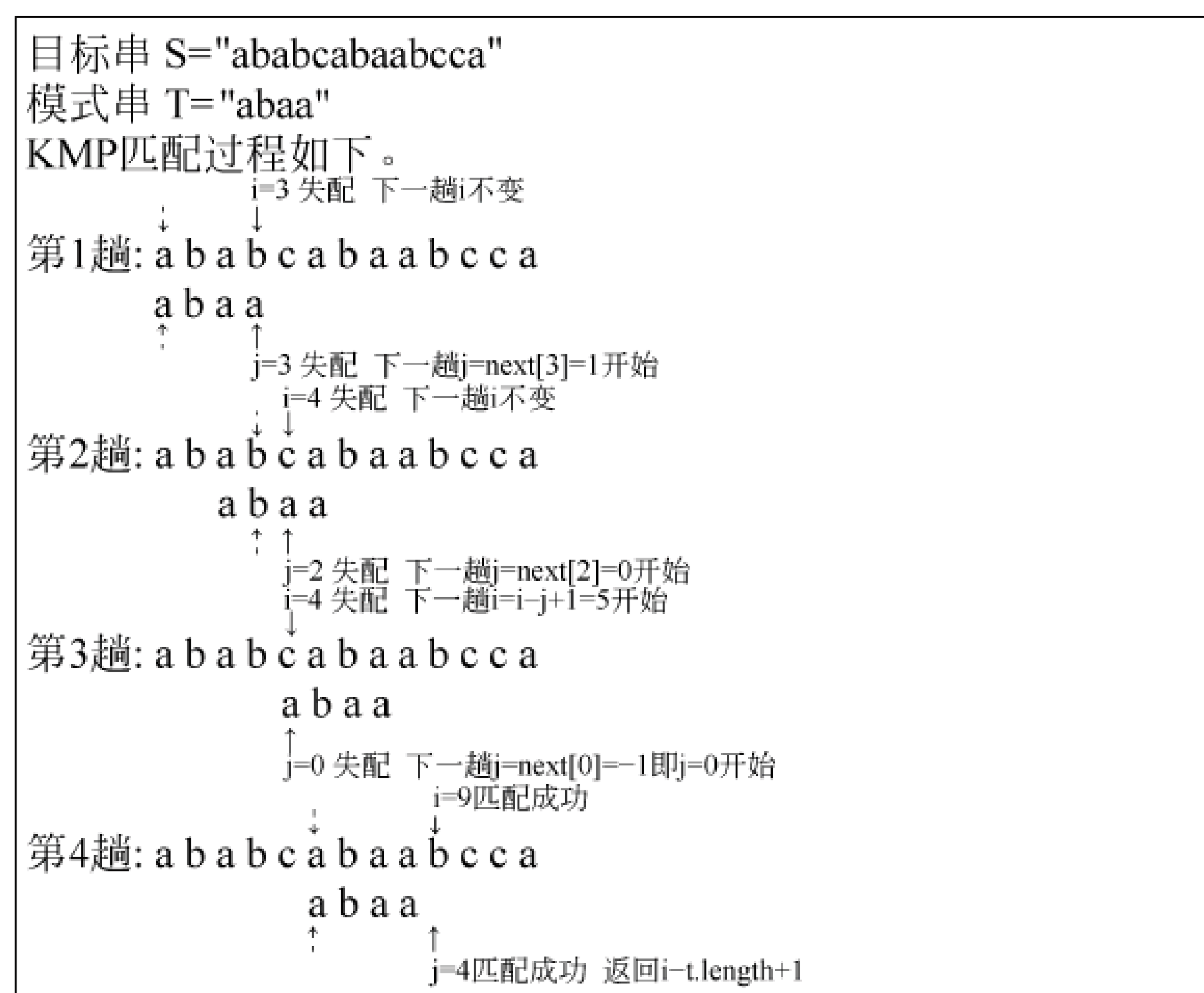


图 4.9 KMP 算法匹配过程

显然,KMP 算法是在已知模式串的 next[ ]数组的基础上执行的,那如何实现计算 next[ ]数组呢?首先根据图 4.8 的分析过程得出计算该数组的递推模型。

当 j=0 时,根据 next 函数的定义可知 next[j]=-1;

设存在 next[j]=k,即在"$t_0t_1\cdots t_{k-1}$"="$t_{j-k}t_{j-k+1}\cdots t_{j-1}$"时,k 是满足该等式的最大值。next[j+1]的值有以下两种情况。

- 若 $t_k=t_j$,即在模式串中有

"$t_0 t_1 \cdots t_{k-1} t_k$" = "$t_{j-k} t_{j-k+1} \cdots t_{j-1} t_j$"

存在，且不可能存在某个 $k'>k$ 也满足上式，则式

$$next[j+1]=next[j]+1=k+1$$

成立。

- 如果 $t_k \neq t_j$ 此时可以把求 next[j+1]的值看成如图 4.10 所示的模式匹配问题，即把模式串向右滑动至 $k'=next[k]$ $(0<k'<k<j)$，如果 $t'_k \neq t_j$，则说明在主串 t 中第 j+1 个字符之前存在一个长度为 k′的子串满足

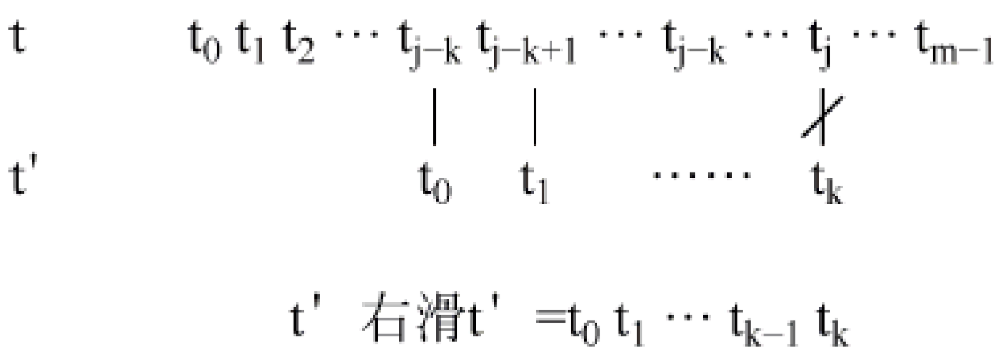


图 4.10　next[j+1]计算过程

"$t_0 t_1 \cdots t_{k'}$" = "$t_{j-k'} t_{j-k'+1} \cdots t_j$"

即下式：

$$next[j+1]=k+1=next[k]+1$$

成立。

此时，如果仍有 $t_k \neq t_j$，那么将模式串 t′继续向右滑动到 $k''=next[k']$。以此类推，直到某次匹配成功或匹配失败。设第 k′向右滑动匹配失败，则有 $next[k'^{-1}]=-1$，所以下式

$$next[j+1]=next[k'^{-1}]+1=-1+1=0$$

成立。

【例 4.3】　求模式串 T="abaabcac"的 next[j]值。求解过程如下。

初始化：

$$j=0, k=-1, next[0]=-1;$$

第 1 趟：

$$j=1, k=0, next[1]=0;$$

第 2 趟：

$$k=next[0]=-1, j=1$$

第 3 趟：

$$j=2, k=0, next[2]=0$$

第 4 趟：

$$j=3, k=1, next[3]=1$$

第 5 趟：

$$k=next[1]=0, j=3$$

第 6 趟：

$$j=4, k=1, next[4]=1$$

第 7 趟：

$$j=5, k=2, next[5]=2$$

第 8 趟：

$$k=next[2]=0, j=5$$

第 9 趟：

$$k=next[0]=-1, j=5$$

第 10 趟：

$$j=6, k=0, next[6]=0$$

第 11 趟：

$$j=7, k=1, next[7]=1$$

KMP 算法关键在于对 next[ ]数组的计算，下面给出 next[ ]的计算过程。

```
void  get_ next(SqString T,int &next[ ])
{//求模式串 T 的 next 函数值并存入数组 next
int j=0,k=-1;
next[0]=-1;
while(j<T.length)
{
if(k==-1|| T.data[j]==T.data[k])
 {++j;++k;next[j]=k;}
else
  k=next[k];                                 //下次比较新的 k 值
}
}
```

本算法的运行时间取决于 while( )循环，因此算法的时间复杂度是 O(T.length)。

模式串 T 的 next[ ]数组计算完成后，如何使用该数组进行快速匹配，通过下面的算法实现串的匹配过程。

```
int Index_KMP(SqString S,SqString T,int next[ ])
{
//利用模式串 T 的 next 函数求 T 在目标串 S 中首次出现的位置
i=0;j=0;
while(i<S.length &&j<T.length)
{
if(j==-1||S.data[i]==T.data[j])
  {++i;++j;}                                 //继续比较后续字符
else
   j=next[j];                                //模式串向右移动
    }
if(j>=T.length)                              //匹配成功
  return i-T.length+1;
else
  return -1;
}
```

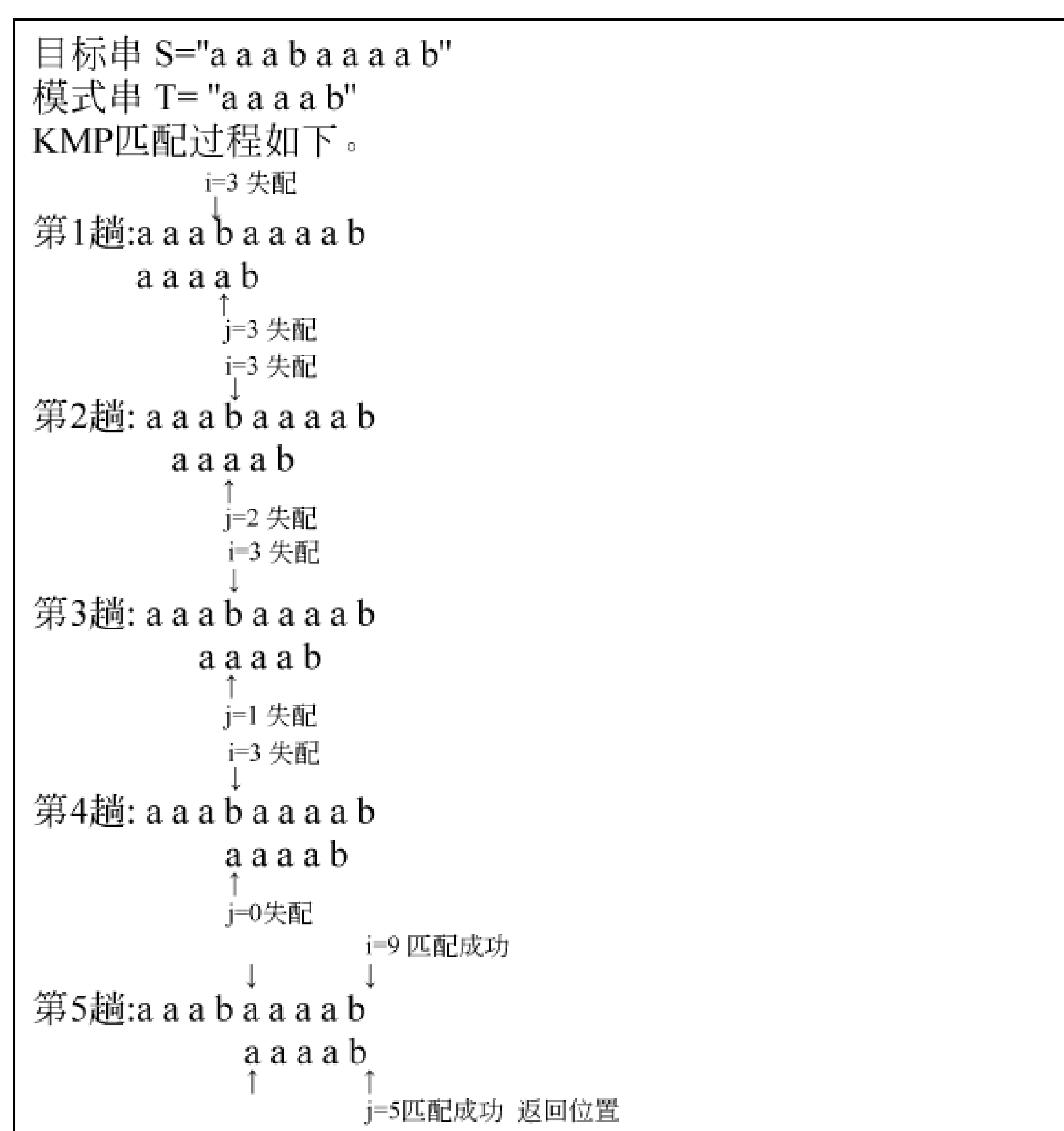
该算法的运行时间取决于 while 循环。由于算法中无回溯现象，在进行相应的字符比较后，要么下标 i 在 S 串中后移一位(加上 1)，要么是调整模式串的下标值，继续向后比较。字符比较的次数最多为 O(S.length)。

结合上述两个算法可知，如果目标串的长度为 n，模式串的长度为 m，则包括 next[ ]的

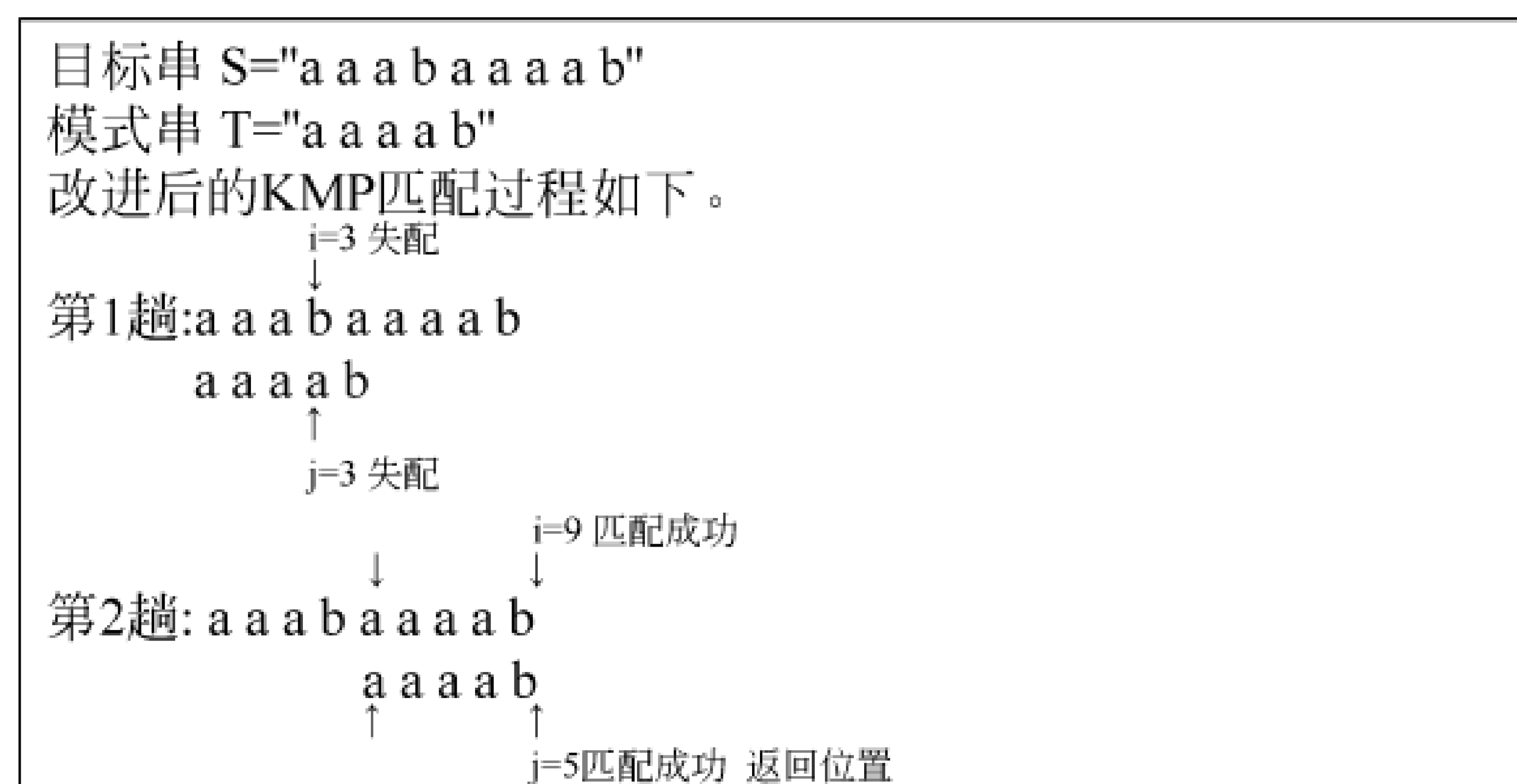
计算过程,整个 KMP 算法的时间复杂度为 O(n+m)。

前面定义的 next 函数虽然较古典匹配算法来说,效率上改进十分大,但是在某些情况下尚有缺陷。如果模式串 T="a a a a b"在和目标串 S="a a a b a a a a b"匹配时,其匹配过程如图 4.11 所示。

图 4.11(a)中,i=3,j=3 时,$s_i \neq t_j$,即对应的'b'≠'a'。由 next 函数的定义可知,i 不变,j=next[3]=2 重新开始匹配,而此时的t_2 仍然为'a',于是新一趟匹配开始,只比较一次就失败;继续回溯下标 j=next[2]=1 重新开始匹配,而此时的 t_1 仍然为'a',于是新一趟匹配开始,只比较一次就宣告失败;继续回溯下标 j=next[1]=0 重新开始,但此时的t_0 仍然是'a',重复上述步骤,直到 i 后移一位,j=0,重新开始匹配,直到成功。那么,此类情况下的 KMP 匹配和简单匹配并无差别,时间效率同样不高,于是,对于 KMP 匹配过程,可适当提出改进措施,如图 4.11(b)所示。



(a) 改进KMP之前的匹配过程



(b) 改进KMP之后的匹配过程

图 4.11 KMP 改进前后的匹配过程对比

由以上思想对 next 函数进行改进,得到 nextval 函数如下。

j 模式串	0 a	1 a	2 a	3 a	4 b
next[j]	−1	0	1	2	3
nextval[j]	−1	−1	−1	−1	3

nextval 函数的计算过程描述如下。

```
void get_ nextval(SqString T,int nextval[ ])
{//求模式串 T 的 next 函数修正值并存入数组 nextval
i=0;
nextval[0]=-1;
j=0;
while(i<T.length)
{
  if(j==0||T.data[i]==T.data[j])
 {
  ++i;++j;
  if (T.data[i]!=T.data[j])
      nextval[i]=j;
  else
      nextval[i]=nextval[j];
  }
 else
     j=nextval[j];
 }
}
```

4.4 综合案例

4.4.1 文本编辑

文本编辑程序是一个面向用户的系统服务程序,广泛用于源程序的输入和修改,甚至用于报刊和书籍的编辑排版以及办公室的公文书信的起草和润色。文本编辑的实质是修改字符数据的形式或格式。虽然各种文本编辑程序的功能强弱不同,但是其基本操作是一致的,一般都包括串的查找、插入和删除等基本操作。

为了编辑的方便,用户可以利用换页符和换行符把文本划分为若干页,每页有若干行(当然,也可不分页,而把文件直接划成若干行)。我们可以把文本看成是一个字符串,称为文本串。页则是文本串的子串,行又是页的子串。

例如,有下列一段源程序:

```
main(){
float a,b,max;
```

```
scanf("%f,%f",&a,&b);
if (a>b)   max=a;
else       max=b;
     }
```

我们可以把此程序看成是一个文本串。将该文本串输入内存后见表 4.1。图中，↙为换行符。

表 4.1　文本格式示例

200	m	a	i	n	(	)	{	↙	f	l	o	a	t		a	,	b	,	m	a	x	;
222	↙	s	c	a	n	f	(	“	%	f	,	%	f	”	,	&	a	,	&	b	)	;
244	↙	i	f	(	a	>	b	)		m	a	x	=	a	;	↙	e	l	s	e		m
266	a	x	=	b	;	↙	}	↙														

为了管理文本串的页和行，在进入文本编辑的时候，编辑程序先为文本串建立相应的页表和行表，即建立各子串的存储映像。页表的每一项给出了页号和该页的起始行号。行表的每一项则指示每一行的行号、起始地址和该行子串的长度。假设表 4.1 所示文本串只占一页，且起始行号为 100，则该文本串的行表见表 4.2。

表 4.2　文本行表及信息排列

行　　号	起 始 地 址	长　　度
100	200	8
101	208	15
102	223	22
103	245	15
104	260	12
105	272	2

文本编辑程序中的页指针、行指针和字符指针，分别指示当前操作的页、行和字符。如果在某行内插入或删除若干字符，则要修改行表中该行的长度。若该行的长度超出了分配给它的存储空间，则要为该行重新分配存储空间，同时还要修改该行的起始位置。如果要插入或删除一行，就要涉及行表的插入或删除。若被删除的行是所在页的起始行，则还要修改页表中相应页的起始行号(修改为下一行的行号)。为了查找方便，行表是按行号递增顺序存储的，因此，对行表进行的插入或删除运算，需移动操作位置以后的全部表项。页表的维护与行表类似，在此不再赘述。由于访问是以页表和行表作为索引的，所以在作行和页的删除操作时，可以只对行表和页表作相应的修改，不必删除所涉及的字符，这可以节省不少时间。

4.4.2　建立词索引表

信息检索是计算机应用的重要领域之一。由于信息检索的主要操作是在大量的存放在磁盘上的信息中查询一个特定的信息，为了提高查询效率，一个重要的问题是建立一个好的索引系统。例如，我们在第 1 章中提过的图书馆书目检索系统中有 3 张索引表，分别可按书

名、作者名和分类号编排。在实际系统中,按书名检索并不方便,因为很多内容相似的书籍其书名不一定相同。因此,较好的办法是建立"书名关键词索引"。

例如,与表4.3中书目对应的关键词索引表见表4.4,很容易从关键词索引表中查询到他所感兴趣的书目。为了便于查询,可将此索引表设定为按词典有序的线性表。

表4.3 书目文件

书　号	书　　名	书　号	书　　名
005	Algorithmic Design Analysis	034	Computer Network
010	Database System	050	Software Engineering
023	Computer Data Structures	067	Numerical Analysis

表4.4 关键词索引表

关　键　词	书号索引	关键词	书号索引
algorithmic	005	engineering	050
analysis	005,067	network	034
computer	023,034	numerical	067
data	010,023	software	050
database	010	structures	023
design	005	system	010

下面要讨论的是如何从书目文件生成这个有序词表。

重复下列操作,直至文件结束。

step1:从书目文件中读入一个书目串。

step2:从书目串中提取所有关键词插入词表。

step3:对词表中的每一个关键词,在索引表中进行查找,并作相应的插入操作。

为识别从书名串中分离出来的单词是否是关键词,需要一张常用词表(英文书名中的"常用词"指的是诸如an、a、of、the等词)。顺序扫描书名串,首先分离单词,然后查找常用词表,若不和表中的任一词相等,则为关键词,插入临时存放关键词的词表中。

在索引表中查询关键词时,可能出现两种情况:其一是索引表上已有此关键词的索引项,只要在该项中插入书号索引即可;其二是需在索引表中插入此关键词的索引项,插入应按字典有序原则进行。下面重点讨论第三个操作的具体实现。

首先设定数据结构。

词表为线性表,只存放一本书的书名中的若干关键词,其数量有限,则采用顺序存储结构即可,其中每个词是一个字符串。

索引表为有序表,虽是动态生成,在生成过程中需频繁进行插入操作,但考虑索引表主要为查找用,为了提高查找效率(采用第8章中将讨论的折半查找法),宜采用顺序存储结构;表中每个索引项都包含两个内容:其一是关键词,因索引表为常驻结构,所以应考虑节省存储,采用堆分配存储表示的串类型;其二是书号索引,由于书号索引是在索引表的生成过程中逐个插入的,且不同关键词的书号索引个数不等,甚至可能相差很多,所以宜采用链表结构的线性表。

```
# define   MaxBookNum   1000                    //假设只对 1000 本书创建索引表
#define    MaxKeyNum    2500                    //索引表的最大容量
#define MaxLineLen      500                     //书目串的最大长度
#define MaxWordNum    10                        //词表的最大容量
typedef   struct {
   char    * item[ ];                           //字符串的数组
   int     last ;                               //词表的长度
} WordListType ;                                //词表类型(顺序表)
typedef int elemtype;                           //定义链表的数据元素类型为整型(书号类型)
typedef   struct {
   HStringkey;                                  //关键词
   LinkList bnolist;                            //存放书号索引的链表
}IdxTermType ;                                  //索引项类型
typedef   struct{
   IdxTermType      item[MaxKeyNum+1];
   int       last;
}IdxListType;                                   //索引表类型(有序表)
   //主要变量
char  *  buf;                                   //书目串缓冲区
WordListType   wdlist;                          //词表
   //基本操作
void InitIdxList(IdxListType &idxlist);
   //初始化操作,置索引表 idxlist 为空表,且在 idxlist. item[0]设一空串
void GetLine (FILE   f);
   //从文件 f 中读入一个书目信息到书目串缓冲区 buf
void ExtractKeyWord(elemtype &bno);
   //从 buf 中提取书名关键词到词表 wdlist,将书号存入 bno
Status InsLdxl,ist(IdxListType &idxlist,elemtype bno);
   //将书号为 bno 的书名关键词按词典顺序插入索引表 idxlist
void PutText(FILE g,IdxListType idxlist);
   //将生成的索引表 idxlist 输出到文件 g
void main() {                                   //主函数
   if (f = openf("Booklnfo.txt","r"))
      if (g = openf("Bookldx.txt","w")){
           InitIdxList(idxlist);                //初始化索引表 idxlist 为空表
           while (!feof(f)) {
            GetLine(f);                         //从文件 f 中读入一个书目信息到 buf
            ExtractKeyWord(BookNo);             //从buf 中提取关键词到词表,将书号存入 BookNo
            InsIdxList(idxlist,BookNo);         //将书号为 BookNo 的关键词插入索引表
             }
     PutText(g,idxlist);                        //将生成的索引表 idxlist 输出到文件 g
   }
}//main
```

为在索引表上进行插入操作,要先实现下列操作。

```
voidGetWord(int i,HString &wd);
   //用 wd 返回词表 wdlist 中的第 i 个关键词
int Locate(IdxListTypeidxlist,HString wd,Boolean &b);
//在索引表 idxlist 中查询是否存在与 wd 相等的关键词,若存在,则返回其在索引表
//中的位置,且 b 取值 TRUE;否则返回插入位置,且 b 取值 FALSE
```

```
void InsertNewKey(IdxListType &idxlist,int i,HString wd);
  //在索引表 idxlist 的第 i 项上插入新关键词 wd,并初始化书号索引的链表为空表
Status InsertBook(IdxListType &idxlist,int i,intbno);
  //在索引表 idxlist 的第 i 项中插入书号为 bno 的索引
```

由此可得索引表的插入算法如 InsertIdxList()所示

```
Status InsertIdxList(IdxListType &idxlist,int bno){
  for (i=0;i<wdlist.last;++i){
  GetWord(i,wd);     j=Locate(idxlist, wd,b);
  if(!b)   InsertNewKey(idxlist,j,wd);                //插入新的索引项
  return   (InsertBook(idxlist,j,bno));               //插入书号索引
  }
}  //InsertIdxList
```

其中 4 个操作的具体实现分别如下面的 4 个算法所示。

```
void GetWord(int i,HString &wd){
  p= * (wdlist.item+i);                              //取词表中的第 i 个字符串
  StrAssign(wd,p);                                   //生成关键字字符串
  }//GetWord
int Locate(IdxListType &idxlist,HString wd,Boolean &b) {
        for ( i = idxlist.last-1;
          (m = StrCompare(idxlist.item[i].key,wd)) >0; --i);
        if (m==0)   {b = TRUE;return i;   }          //找到
        else {b = FALSE;      return i+1;   }
}//Locate
void InsertNewKey(int i,StfType wd){
  for(j=idxlist.last-1;j>=i;--j)                     //后移索引项
      idxlist.item[j+1]=idxlist.item[j];
  //插入新的索引项
  StrCopy(idxlist.item[i]. key,wd);                  //串赋值
  InitList(idxlist.item[i]. bnolist);                //初始化书号索引表为空表
  ++ idxlist. Last;
}//InsertNewKey
Status InsertBook(IdxListType &idxlist,init,int bno){
if(!MakeNode(p,bno)   return   OVERFLOW;             //分配失败
Append(idxlist. item[i]. bnolist,p);                 //插入新的书号索引
return OK;
}//InsertBook
```

本 章 小 结

本章主要介绍了串的基本知识,主要学习要点如下。

- 理解串的特殊性,掌握串的定义和相关概念。
- 掌握串的存储结构及类型定义。
- 掌握串的模式匹配算法(简单匹配算法和快速匹配算法)的实现过程。
- 了解串的应用实例。

第5章 数组和广义表

前几章讨论的线性结构中的数据元素都是非结构的原子类型,原子类型的数据元素是不能再分解的。本章讨论的两种数据结构(**数组**和**广义表**)与之前讨论的数据结构有所不同,这两种数据结构均可以被看成是线性表在下述含义上的扩展,即表中的数据元素本身也是一个数据结构。

数组是C/C++中常见的数据类型,几乎所有的程序设计语言都把数组类型设定为固有类型。本章以抽象数据类型的形式讨论数组的定义和实现,便于加深对数组类型的理解。

5.1 数组的定义及抽象数据类型

5.1.1 数组的定义

数组是由多个类型相同的数据元素组成的一个有限序列。在定义上,数组与线性表的形式几乎一致。从逻辑结构上看,数组A是由$n(n>1)$个相同类型数据元素$a_1,a_2,\cdots,a_n$构成的有限序列,其逻辑表示为

$$A=(a_1,a_2,\cdots,a_n)$$

其中,$a_i(1\leqslant i\leqslant n)$表示数组A的第i个元素。

一个二维数组可以被看作是一个特殊的一维数组,即每个数组元素都是相同类型的一维数组。以此类推,任何多维数组都可以被看成是一个线性表,且表中的每个数据元素也是一个线性表。多维数组是线性表的推广。

推广到$d(d\geqslant 3)$维数组,不妨把它看作是一个以$d-1$维数组作为数据元素的线性表;或者这样理解,它是一种较复杂的线性表结构,由简单的数据结构(即线性表)辗转合成而得。

5.1.2 数组的抽象数据类型

抽象数据类型d维数组的定义如下。

```
ADT Array{
  数据对象:
  D={a_{j1,j2,…,jd} | j_i=1,…,b_i,i=1,2,…,d}          //第i维的长度为b_i
  数据关系:
  R={r_1,r_2,…,r_d}
```

$r_i=\{<a_{j1\cdots ji\cdots jd}, a_{j1\cdots ji+1\cdots jd}> \mid 1\leqslant j_k\leqslant b_k, 1\leqslant k\leqslant d$ 且 $k\neq i, 1\leqslant j_i\leqslant b_{i-1}, i=2,\cdots,d\}$

基本运算:

Value(A, &e, $index_1,\cdots,index_d$)

初始条件:A 是 d 维数组,e 为元素变量,随后是 d 个下标值。

操作结果:若各下标不超界,则 e 赋值为所指定的 A 的元素值,并返回 OK。

Assign(&A, e, $index_1,\cdots,index_n$)

初始条件:A 是 d 维数组,e 为元素变量,随后是 d 个下标值。

操作结果:若下标不超界,则将 e 的值赋给所指定的 A 的元素,并返回 OK。

ADisp(A, $b_1, b_2, \cdots, b_d$)

初始条件:A 是 d 维数组,随后是 d 个下标值。

操作结果:输出 d 维数组 A 的所有元素值。

} ADT Array

数组一般不进行插入和删除操作。通常,数组被建立以后,元素的个数和元素之间的关系不再发生变化。这个特点使得数组不像线性表那样,可以在表中的任意位置上进行插入和删除元素。数组中常见的操作是:给定下标读取数组元素的值和修改数组元素的值。因此,除了使用下标,也可以通过计算数组元素的地址实现上述操作。

5.2 数组的顺序存储与寻址

从存储结构上看,数组的所有元素都存储在一个地址连续的内存单元中。几乎所有的计算机语言都支持数组类型,以 C/C++语言为例,数组数据类型具有以下性质。

- 数组一经定义,数组中元素的个数不能随意增减。
- 数组中的数据元素的类型必须相同。
- 数组中的每个数据元素都对应一个唯一的下标值。
- 数组是一种随机的存储结构,可随机存取数组中的任意数据元素。

正是由于数组的所有元素都存储在连续的内存单元中,所以线性表的顺序存储结构也应采用一维数组描述。

在一维数组中,一旦 a_1 的存储地址 $LOC(a_1)$ 确定,且假设每个数据元素占用 k 个存储单元,则任一数据元素 a_i 的存储地址 $LOC(a_i)$ 均可由式(5.1)求出。

$$LOC(a_i)=LOC(a_1)+(i-1)\times k \qquad (1\leqslant i\leqslant n) \tag{5.1}$$

该式说明,一维数组中的任一数据元素的存储地址都可直接计算得到,即一维数组中任一数据元素都可直接存取。正因如此,所以一维数组具有随机存储特性。同样,二维及多维数组也具有随机存储特性。一个 m 行 n 列的二维数组 $A_{m\times n}$ 如图 5.1 所示。

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$

图 5.1 二维数组 $A_{m\times n}$

从逻辑结构上看,二维数组 A 中的元素构成 m 行 n 列的行列式。数组中任意的元素 a_{ij} 位于数组的第 i 行、第 j 列。观察 a_{ij} 所处的行发现,第 i 行的 n 个元素可形成线性表序列。观察 a_{ij} 所处的列发现,第 j 列的 m 个元素也可以形成线性表序列。因此,对二维数组 A 中的元素按行或按列抽象可得到如图 5.2 所示的两种线性表结构。

按行抽象:假设

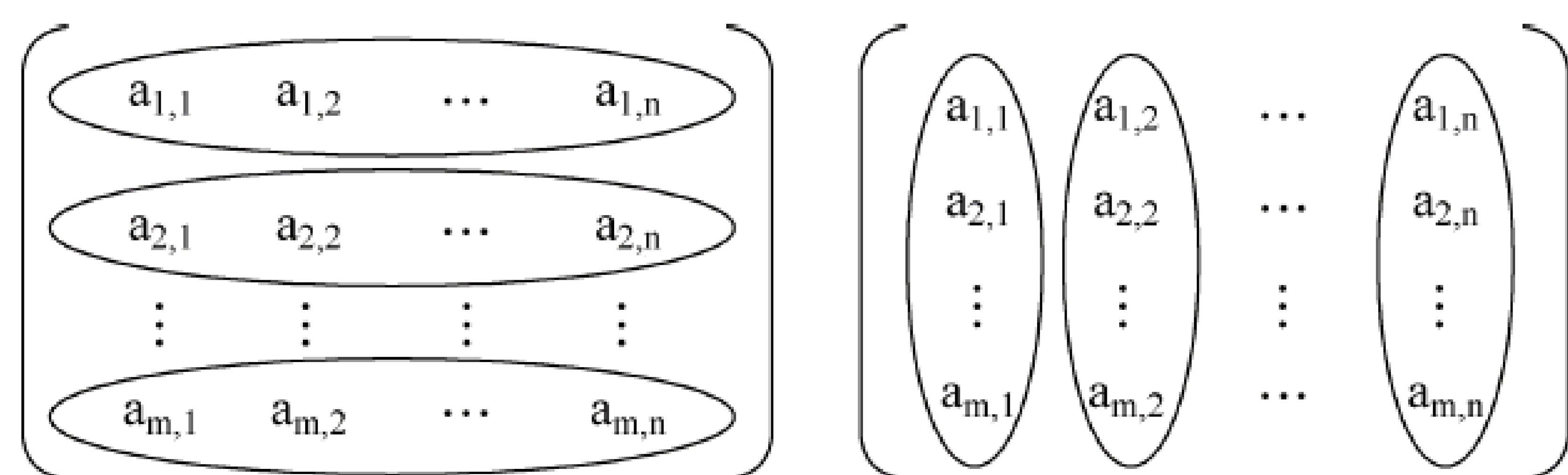


图 5.2　二维数组 $A_{m\times n}$按行或按列抽象

$$\alpha_1=[a_{1,1},a_{1,2},a_{1,3},\cdots,a_{1,n}]$$
$$\alpha_2=[a_{2,1},a_{2,2},a_{2,3},\cdots,a_{2,n}]$$
$$\vdots$$
$$\alpha_m=[a_{m,1},a_{m,2},a_{m,3},\cdots,a_{m,n}]$$

于是,二维数组 A 抽象成长度为 m 的线性表 La=($\alpha_1,\alpha_2,\cdots,\alpha_m$)。其中,$\alpha_i$为线性表。

按列抽象:假设

$$\beta_1=[a_{1,1},a_{2,1},a_{3,1},\cdots,a_{m,1}]$$
$$\beta_2=[a_{1,2},a_{2,2},a_{3,2},\cdots,a_{m,2}]$$
$$\vdots$$
$$\beta_n=[a_{1,n},a_{2,n},a_{3,n},\cdots,a_{m,n}]$$

于是,二维数组 A 抽象成长度为 n 的线性表 Lb=($\beta_1,\beta_2,\cdots,\beta_n$)。其中,$\beta_i$为线性表。

因此,二维数组 A 可以看成是以线性表为数据元素的线性表。同样,一个 n 维数组也可以抽象成以 n−1 维数组为数组元素的一维数组。

对于二维数组来说,由于计算机的存储结构是线性的,如何用线性的存储结构存放二维数组元素?这就有一个按行或按列排放次序问题。于是,对图 5.1 中 m 行 n 列的二维数组 $A_{m\times n}$可以按两种方式进行顺序存放与寻址。



视频讲解

5.2.1　以行序为主序

在 C、Pascal、Basic 等大多数程序设计语言中采用的是以行序为主序的存储方式,即先存储第一行上的数据元素,然后存储第二行上的数据元素,直到最后一行元素被存储完毕。图 5.2 按行将二维数组 A 抽象成线性表 La=($\alpha_1,\alpha_2,\cdots,\alpha_m$)。对线性表 La 进行顺序存储,可得到如图 5.3所示的存储结构。



图 5.3　以行序为主序顺序存储

对一个已知以行序为主序的计算机系统,当二维数组第一个数据元素 $a_{1,1}$的存储地址 LOC($a_{1,1}$)和每个数据元素占用的存储单元 t 确定后,该二维数组中任一数据元素 $a_{i,j}$的存储地址可由式(5.2)确定。

$$LOC(a_{i,j})=LOC(a_{1,1})+[(i-1)\times n+(j-1)]\times t \tag{5.2}$$

其中，$[(i-1)\times n+(j-1)]$表示按行存储时 $a_{i,j}$之前存放的元素个数。

推广到一般情况，就可以得到 n 维数组的数据元素存储地址的计算公式为

$$LOC[j_1,j_2,\cdots,j_n]=LOC[0,0,\cdots,0]+(b_2b_3\cdots b_nj_1+b_3b_4\cdots j_2+b_nj_{n-1}+j_n)\times L$$

$$=LOC[0,0,\cdots,0]+\left(\sum_{i=1}^{n-1}j_i\prod_{k=i+1}^{n}b_k+j_n\right)\times L$$

其中，LOC[0,0,…,0]是 $a_{0,0,\cdots,0}$的存储地址；b_i是数组第 i 维的长度。

【例 5.1】 在 C++语言中定义了数组 a[5][6]，每个元素占 8 个字节，假定该数组的首地址为 20000H，请计算数组元素 a[3][4]的字节地址。

解：由于 C/C++中数组的行、列下标均为 0，且默认以行为主序存储，元素 a[3][4]位于第 4 行第 5 列，根据式(5.2)可知

$LOC(a_{3,4})=LOC(a_{0,0})+(i\times n+j)\times t=20000+(3\times6+4)\times8=20176$。



视频讲解

5.2.2 以列序为主序

在 FORTRAN 等少数程序设计语言中采用的是以列序为主序的存储方式，即先存储第一列的数据元素，紧接着存储第二列数据元素，最后存储第 n 列数据元素。图 5.2 按列将二维数组 A 抽象成线性表 $Lb=(\beta_1,\beta_2,\cdots,\beta_n)$。对线性表 Lb 进行顺序存储可得到如图 5.4 所示的存储结构。



图 5.4 以列序为主序顺序存储

对一个已知以列序为主序的计算机系统，当二维数组第一个数据元素 $a_{1,1}$的存储地址 $LOC(a_{1,1})$和每个数据元素占用的存储单元 t 确定后，该二维数组中任一数据元素 $a_{i,j}$的存储地址可由式(5.3)确定。

$$LOC(a_{i,j})=LOC(a_{1,1})+[(j-1)\times m+(i-1)]\times t \tag{5.3}$$

其中，$[(j-1)\times m+(i-1)]$表示按列存储时 $a_{i,j}$之前存放的元素个数。

【例 5.2】 假设某二维数组 A[m][m]按行存储时元素 $a_{i,j}$的计算公式如下：

$$LOC(a_{i,j})=LOC(a_{0,0})+(i\times m+j)\times t$$

请列出该数组按列存储时元素 $a_{i,j}$的计算公式。

解：根据按行存储的公式可知，二维数组 A 的行号和列号均从 0 开始，即 $a_{0,0}$为第一个元素，每个元素占 t 个存储单元，元素 $a_{i,j}$在数组中的位置是第 i+1 行、第 j+1 列，由式(5.3)可知：

按列存储时元素 $a_{i,j}$的计算公式：$LOC(a_{i,j})=LOC(a_{0,0})+(j\times m+i)\times t$。

【例 5.3】 求解约瑟夫问题：设有 n 个人站成一圈，其编号为 1～n，从编号为 1 的人开始按顺时针方向“1,2,3,4,…”循环报数，数到 m 的人出列，然后从出列者的下一个人开始重新报数，数到 m 的人又出列，如此重复进行，直到 n 个人都出列为止。要求输出这 n 个人的出列顺序。

例如，有 8 个人的初始序列为

1 2 3 4 5 6 7 8

当 $m=4$ 时，出列顺序为

4 8 5 2 1 3 7 6

分析：采用一维数组 p[ ]存放人的编号，即先将 n 个人的编号存到 p[0]～p[n－1]中。从编号为 1 的人(下标 t=0)开始循环报数，数到 m 的人(下标 t=(t+m－1)%i)出列，输出 p[t]并将其从数组中删除(即将后面的元素前移一个位置)。因此，每次报数的起始位置就是上次报数的出列位置。反复执行，直到出列 n 个人为止。算法如下。

```
void josephus(int n,int m)
{
int p[MaxSize];
int i,j,t;
for(i=0;i<n;i++)                          //构建初始序列
    p[i]=i+1;
t=0;                                      //首次报数的起始位置
printf("出列顺序");
for(i=n;i>=1;i--)                         //i 为数组 p 中的人数
{ t=(t+m-1)%i;                            //t 为出列者的编号
  printf("%d",p[t]);                      //编号为 t 的元素出列
  for(j=t+1;j<=i-1;j++)                   //后面的元素前移一个位置
     p[j-1]=p[j];
  }
printf("\n");
}
```

5.3 特殊矩阵及其压缩存储

特殊矩阵是指非零元素或零元素的分布有一定规律的矩阵。为了节省存储空间，特别对于高阶矩阵，可以利用特殊矩阵的规律，对它们进行压缩存储。也就是说，使多个相同的非零元素共享同一个存储单元，对零元素则不分配存储空间。特殊矩阵的主要形式有对称矩阵、对角矩阵等。本节研究的均是方阵，即行数和列数相同的矩阵。

5.3.1 对称矩阵



视频讲解

若一个 n 阶方阵 A[n][n]中的元素满足 $a_{i,j}=a_{j,i}$ $(1\leqslant i,j\leqslant n)$，则称其为 n 阶**对称矩阵**。

由于对称矩阵中的元素关于主对角线对称，压缩存储时使得对称的元素共享一个存储空间，即为每一对对称元素只分配一个存储空间，于是只需存储对称矩阵中上三角或下三角中的元素。可以将 n^2 个元素压缩存储到 $n(n+1)/2$ 个元素的空间中，如图 5.5 所示。

假设以一维数组 B[0..n(n+1)/2－1]作为 n 阶对称矩阵 A 的存储结构，在 B 中只存储对称矩阵 A 的下三角元素 $a_{i,j}$ $(i\geqslant j)$。若按行存储时，其存储结构如图 5.6 所示。

假设 A 的下三角中的元素 $a_{i,j}$ 存储在 B 数组中。于是，下三角矩阵中的元素 $a_{i,j}$ 和 B[k]

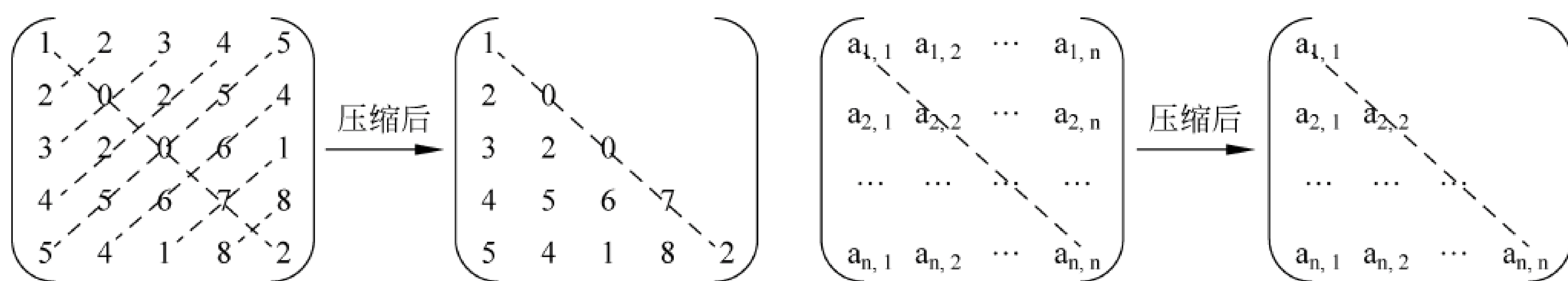


图 5.5　n 阶对称矩阵的压缩

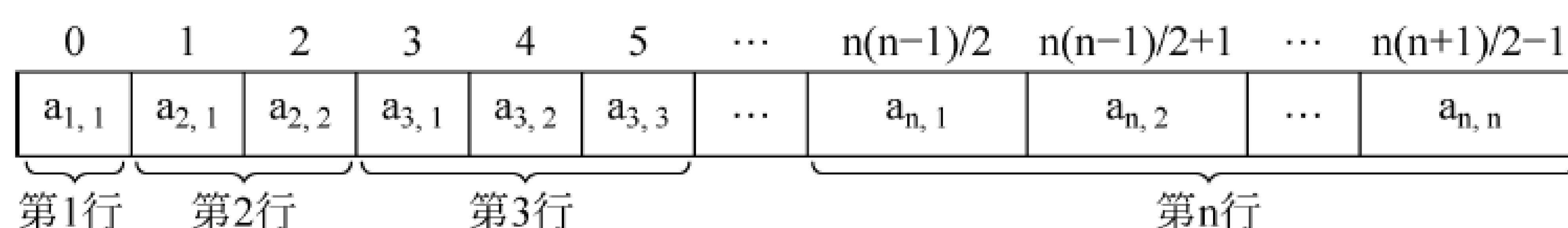


图 5.6　对称矩阵的压缩存储

之间存在一一对应关系,如式(5.4)所示。

$$k=\begin{cases}\dfrac{i(i-1)}{2}+(j-1) & i\geqslant j\text{ 时}\\ \dfrac{j(j-1)}{2}+(i-1) & i\leqslant j\text{ 时}\end{cases}\tag{5.4}$$

5.3.2　下(上)三角矩阵



有些非对称的矩阵也可借用此方法存储,如图 5.7 所示为 n 阶下三角矩阵的压缩。**n 阶下(上)三角矩阵**是指矩阵的上(下)三角(不包括对角线)中的元素均为常数 c 或 0 的 n 阶方阵。下面以 n 阶下三角矩阵为例,讲解三角矩阵的压缩过程。

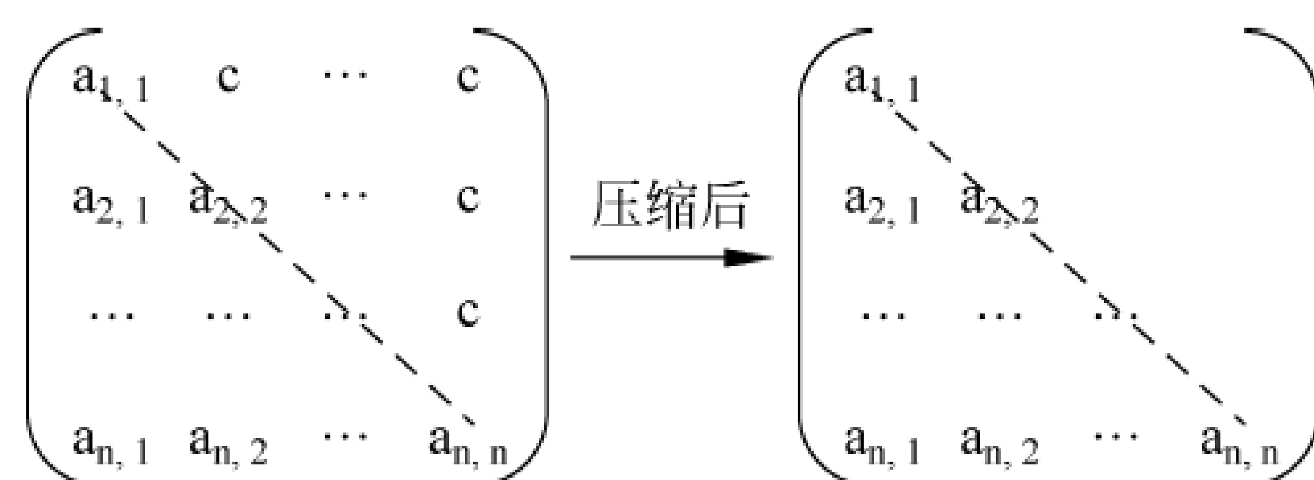


图 5.7　n 阶下三角矩阵的压缩

设以一维数组 B[0..n(n+1)/2]作为 n 阶下三角矩阵 A 的存储结构,若按行存储,则 A 中的元素如图 5.8 所示。

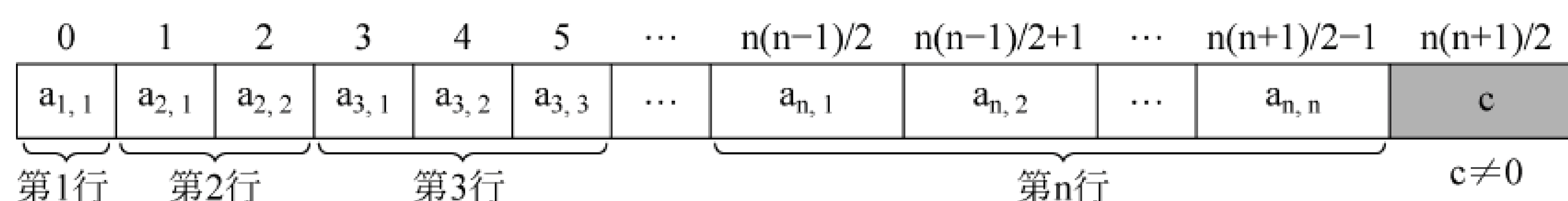


图 5.8　n 阶下三角矩阵的压缩存储

于是,$a_{i,j}$与 B[k]之间的关系如式(5.5)所示。

$$k=\begin{cases}\dfrac{i(i-1)}{2}+(j-1) & i\geqslant j\text{ 时}\\ \dfrac{n(n+1)}{2} & i\leqslant j\text{ 且 }c\neq 0\text{ 时}\end{cases}\tag{5.5}$$

5.3.3 对角矩阵



若一个 n 阶方阵 A 其所有非零元素都集中在以主对角线为中心的带状区域中，则称之为 **n 阶对角矩阵**，即除了主对角线上和直接在对角线上、下方若干条对角线上的元素以外，所有其他的元素均为零。若其主对角线上、下方各有 b 条次对角线，则称 b 为矩阵半带宽，(2b+1)为矩阵带宽。图 5.9 所示为半带宽为 b 的对角矩阵示意图，对这种矩阵，可以以行为主序或以对角线为顺序将其压缩存储到一维数组中。

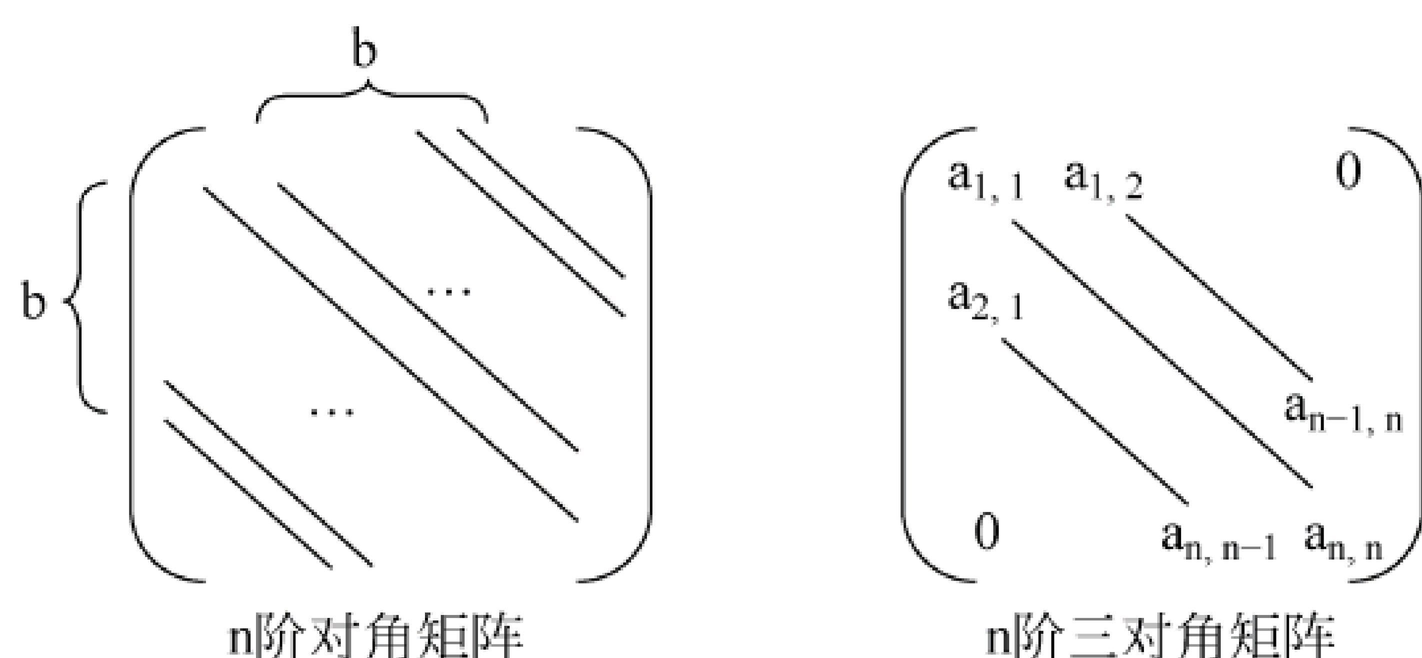


图 5.9 对角矩阵

三对角矩阵只需存储三条对角线上的元素，共 **3n−2** 个元素需要保存。设以一维数组 B[0..3n−3]作为 n 阶三对角矩阵 A 的存储结构，若按行存储，则 A 中的元素如图 5.10 所示。



图 5.10 三对角矩阵的压缩存储

三对角矩阵中元素 $a_{i,j}$ 与 B[k]之间的关系如式(5.6)所示。

$$k=\begin{cases}3i-4+0 & j-i=-1\text{ 时}\\3i-4+1 & j-i=0\text{ 时}\\3i-4+2 & j-i=1\text{ 时}\end{cases}\tag{5.6}$$

对于 b=1 的三对角矩阵，只存储其非零元素，并存储到一维数组 B 中，即将三对角矩阵 A 的非零元素 $a_{i,j}$ 存储到 B 的元素 B[k]中。A 中第 1 行和第 n 行都只有两个非零元素，其余各行均有 3 个非零元素。对于不在第 1 行的非零元素 $a_{i,j}$ 来说，在它前面存储了矩阵的前 i−1 行元素，这些元素的总数为 2+3*(i−2)=3i−4。若 $a_{i,j}$ 是本行中需要存储的第 1 个元素，则 k=2+3*(i−2)+0=3i−4，此时 i=j+1；若 $a_{i,j}$ 是本行中需要存储的第 2 个元素，则 k=2+3*(i−2)+1=3i−3，此时 i=j；若 $a_{i,j}$ 是本行中需要存储的第 3 个元素，则 k=2+3*(i−2)+2=3i−2，此时 i=j−1。

归纳起来，前 i−1 行存储的元素共有 3i−4 个，后面所加的调整值分别有 0、1、2，这与元素 $a_{i,j}$ 所在的位置有关，即与行号 i、列号 j 有关，调整值表达式为(j−i+1)，于是 k=3i−4+(j−i+1)=2i+j−3。

以上讨论的对称矩阵、三角矩阵、对角矩阵的压缩存储方法是把有一定分布规律的值相同的元素(包括 0)压缩存储到一个存储空间中。这样的压缩存储只在算法中按公式做映射即可实现矩阵元素的随机存取。

5.4 稀疏矩阵

实际应用中经常会遇到一类矩阵：其阶数较大，且矩阵中的非零元素个数 s 远小于矩阵元素的总个数 t，即 s≪t，且非零元的分布没有一定规律，通常称该矩阵为**稀疏矩阵**。例如，一个 100×100 的矩阵，若其中只有 100 个非零元素，就可称其为稀疏矩阵。

5.4.1 稀疏矩阵的三元组表示


视频讲解

不同于 5.1 节讨论的几种特殊矩阵的压缩存储方法，稀疏矩阵的压缩存储方法是只存储非零元素，由于稀疏矩阵中非零元素的分布没有任何规律，所以在存储非零元素时还必须同时存储该非零元素对应的行号和列号。这样，稀疏矩阵中的每一个非零元素都需由一个三元组$(i,j,a_{i,j})$唯一确定，稀疏矩阵中的所有非零元素构成三元组线性表。

假设一个 6×7 阶稀疏矩阵 A 中的元素如图 5.11 所示。

以行序为主序扫描图 5.11 所示的稀疏矩阵 A，可得下面的三元组线性表。

```
((1,3,1),(2,3,2),(3,1,3),(4,4,5),(5,5,6),(6,6,7),(6,7,4))
```

观察发现，假设在稀疏矩阵 A 最后一行元素的后面附加整行零，或者在稀疏矩阵 A 最后一列元素后面附加整列零，产生阶数不同的其他稀疏矩阵，但它们却具有相同的非零元素，于是一个三元组线性表可以对应无数个不同的稀疏矩阵，这样在算法设计中会造成很大的麻烦，为了加以区别，在存储三元组表时，附加一个特殊的三元组，即(**行数，列数，非零元个数**)，就使得三元组线性表和稀疏矩阵一一对应起来。

```
((6,7,7),(1,3,1),(2,3,2),(3,1,3),(4,4,5),(5,5,6),(6,6,7),(6,7,4))
```

若把稀疏矩阵的三元组线性表按顺序存储结构存储，则称为稀疏矩阵的三元组顺序表，简称三元组表。三元组表的数据类型定义如下。

```
#define M <稀疏矩阵行数>
#define N <稀疏矩阵列数>
#define MaxSize <稀疏矩阵中非零元素最多个数>
typedef struct{
  int   r,c;                        //非零元的行号和列号
  elemtype   d;                     //元素值
 }TupNode;                          //三元组定义
typedef struct{
int rows;                           //行数
int cols;                           //列数
int nums;                           //非零元素的个数
TupNode   data[MaxSize];
 }TSMatrix;                         //三元组顺序表定义
```

其中，data 域中表示的非零元素通常以行序为主序顺序排列，它是一种下标按行有序的存

储结构，这种有序存储结构可简化大多数稀疏矩阵运算算法。在下面的讨论中，假设 data 域按行有序存储，于是图 5.11 所示稀疏矩阵对应的三元组表如图 5.12 所示。

$$A_{6\times7}=\begin{bmatrix}0&0&1&0&0&0&0\\0&0&2&0&0&0&0\\3&0&0&0&0&0&0\\0&0&0&5&0&0&0\\0&0&0&0&6&0&0\\0&0&0&0&0&7&4\end{bmatrix}$$

图 5.11　稀疏矩阵 $A_{6\times7}$

i	j	e
6	7	7
1	3	1
2	3	2
3	1	3
4	4	5
5	5	6
6	6	7
6	7	4

图 5.12　稀疏矩阵 $A_{6\times7}$ 对应的三元组表

稀疏矩阵运算通常包括矩阵转置、矩阵加、矩阵减、矩阵乘等。这里仅讨论基本运算和矩阵转置运算算法。

1. 对一个二维稀疏矩阵创建其三元组表示

以行序方式扫描二维稀疏矩阵 A，将其非零的元素插到三元组 t 中。

```
void  CreatMat(TSMatrix &t,elemtype A[M][N])
{
int i,j;
t.rows=M;t.cols=N;t.nums=0;
for(i=0;i<M;i++)
   {
for(j=0;j<N;j++)
    if(A[i][j]!=0)
    {
     t.data[t.nums].r=i;data[t.nums].c=j;
     t.data[t.nums].d=A[i][j];t.nums++;
    }
   }
}
```

算法的时间复杂度为 O(M×N)，M 是稀疏矩阵的行数，N 是列数。

2. 三元组元素赋值

对于稀疏矩阵 A，执行 A[i][j]=x。先在三元组 t 中找到适当的位置 k，将 k～t.nums 位置的元素后移一个位置，而后将指定元素 x 插到 t.data[k]处。

```
bool  Value(TSMatrix &t,elemtype x,int i,int j)
{  int k=0,k1;
   if(i>=t.rows||j>=t.cols)
```

```
        return false;                                  //失败时返回 false
    while(k<t.nums&&i>t.data[k].r) k++;               //查找行
    while(k<t.nums&&i==t.data[k].r&&j>t.data[k].c) k++;
                                                       //查找列
    if(t.data[k].r==i&&t.data[k].c==j)                 //存在这样的元素
        t.data[k].d=x;
    else                                               //不存在这样的元素时,插入一个元素
    {   for(k1=t.nums-1;k1>=k;k1--)
        {
        t.data[k1+1].r=t.data[k1].r;
        t.data[k1+1].c=t.data[k1].c;
        t.data[k1+1],d=t.data[k1].d;
            }
        t.data[k].r=i;t.data[k].c=j;t.data[k].d=x;
        t.nums++;
    }
    return true;                                       //成功时返回 true
}
```

3. 将指定位置的元素值赋给变量

对于稀疏矩阵 A,执行 x=A[i][j]。先在三元组 t 中找到指定的位置,再将该处的元素值赋给 x。

```
bool  Assign(TSMatrix t,elemtype &x,int i,int j)
{   int k=0;
    if(i>=t.rows||j>=t.cols)
        return false;                                    //失败时返回 false
    while(k<t.nums&&i>t.data[k].r)k++;                  //查找行
    while(k<t.nums&&i==t.data[k].r&&j>t.data[k].c)k++;
                                                         //查找列
    if(t.data[k].r==i&&t.data[k].c==j)
        x=t.data[k].d;
    else
        x=0;                                             //在三元组中没有找到,表示是零元素
    return true;                                         //成功时返回 true
}
```

4. 输出三元组

从头到尾扫描三元组表 t,依次输出元素值。

```
void  DispMat(TSMatrix t)
{
int i;
if(t.nums<=0)
    return;
printf("\t%d\t%d\t%d\n",t.rows,t.cols,t.nums);
```

```
    printf("\t----------\n");
    for(i=0;i<t.nums;i++)
        printf("\t%d\t%d\t%d\n",t.data[i].r,t.data[i].c,t.data[i].d);
}
```

5. 矩阵转置

对于一个 m×n 的矩阵 $A_{m\times n}$，其转置矩阵是一个 n×m 的矩阵，设为 $B_{n\times m}$，满足 $a_{i,j}=b_{j,i}$，其中 $0\leqslant i\leqslant m-1, 0\leqslant j\leqslant n-1$（参见图 5.13）。

$$A_{6\times 7}=\begin{bmatrix}0&0&1&0&0&0&0\\0&0&2&0&0&0&0\\3&0&0&0&0&0&0\\0&0&0&5&0&0&0\\0&0&0&0&6&0&0\\0&0&0&0&0&7&4\end{bmatrix}\quad B_{7\times 6}=\begin{bmatrix}0&0&3&0&0&0\\0&0&0&0&0&0\\1&2&0&0&0&0\\0&0&0&5&0&0\\0&0&0&0&6&0\\0&0&0&0&0&7\\0&0&0&0&0&4\end{bmatrix}$$

(a) 稀疏矩阵 A 与其转置矩阵 B

i	j	e
6	7	7
1	3	1
2	3	2
3	1	3
4	4	5
5	5	6
6	6	7
6	7	4

i	j	e
7	6	7
1	3	3
3	1	1
3	2	2
4	4	5
5	5	6
6	6	7
7	6	4

(b) 稀疏矩阵 A 与其转置矩阵 B 的三元组表

图 5.13　稀疏矩阵的转置

```
void  TranTat(TSMatrix ta, TSMatrix &tb)
{ int p,q=0,v;                                    //q 为 tb.data 的下标
  tb.rows=ta.cols;tb.cols=ta.rows;tb.nums=ta.nums;
  if(ta.nums!=0)                                  //当存在非零元素时执行转置
  {   for(v=0;v<ta.cols;v++)                      //tb.data[q]中的记录以 c 域的次序排列
        for(p=0;p<ta.nums;p++)                    //p 为 t.data 的下标
          if(ta.data[p].c==v)
          { tb.data[q].r=ta.data[p].c;
            tb.data[q].c=ta.data[p].r;
            tb.data[q].d=ta.data[p].d;
            q++;
```

```
            }
        }
    }
```

以上算法的时间复杂度为 O(ta.cols×t.nums)，而由二维数组存储一个 m 行 n 列矩阵时，其转置算法的时间复杂度为 O(m×n)。最坏情况是当稀疏矩阵中的非零元素个数 ta.nums 和 m×n 同数量级时，上述转置算法的时间复杂度就为 $O(m\times n^2)$。对别的几种矩阵运算也是同样的情况。可见，常规的非稀疏矩阵应采用二维数组存储，只有当矩阵中非零元素个数远小于 m×n 时，才可采用三元组顺序表存储结构。这个结论也同样适用于接下来要讨论的十字链表。

【例 5.4】 请采用三元组存储稀疏矩阵，设计两个稀疏矩阵相加的运算算法。

分析：实现相加运算的两个稀疏矩阵 a 和 b 的行数、列数都必须相同。用 i 和 j 两个变量扫描三元组表 a 和 b，按行序优先方式进行处理，并将结果存放在三元组表 c 中。当 a 的当前元素(简称为 a 元素)和 b 的当前元素(简称为 b 元素)的行号、列号均相等时，将它们的值相加，只有相加值不为 0 时，才在 c 中添加一个新的元素表示相加后的结果。本例算法如下。

```
bool  MatAdd(TSMatrix a, TSMatrix ab, TSMatrix &c)
{  int i=0,j=0,k=0;
   elemtype v;
   if(a.rows!=b.rows||a.cols!=b.cols)
       return false;                          //行数或列数不等时不能进行相加运算
   c.rows=a.rows;c.cols=a.cols;               //c 的行列数与 a 的相同
   while(i<a.nums&&j<b.nums)                  //处理 a 和 b 中的每个元素
   {  if(a.data[i].r==b.data[j].r)            //行号相等时
       { if(a.data[i].c<b.data[j].c)          //a 元素的列号小于 b 元素的列号
          { c.data[k].r=a.data[i].r;          //将 a 元素添加到 c 中
            c.data[k].c=a.data[i].c;
            c.data[k].d=a.data[i].d;
            k++;j++;
              }
         else if(a.data[i].c>b.data[j].c)     //a 元素的列号大于 b 元素的列号
            {c.data[k].r=b.data[j].r;         //将 b 元素添加到 c 中
             c.data[k].c=b.data[j].c;
             c.data[k].d=b.data[j].d;
             k++;j++;
            }
         else                                 //a 元素的列号等于 b 元素的列号
            {v=a.data[i].d+b.data[j].d;
             if(v!=0)                         //只将不为 0 的结果添加到 c 中
            {c.data[k].r=a.data[i].r;
             c.data[k].c=a.data[i].c;
             c.data[k].d=v;
             k++;
```

```
                }
                i++;j++;
                }
    }
    else if(a.data[i].r<b.data[j].r)          //a 元素的行号小于 b 元素的行号
      {   c.data[k].r=a.data[i].r;            //将 a 元素添加到 c 中
          c.data[k].c=a.data[i].c;
                      c.data[k].d=a.data[i].d;
                      k++;i++;
      }
     else                                     //a 元素的行号大于 b 元素的行号
     {   c.data[k].r=b.data[j].r;             //将 b 元素添加到 c 中
        c.data[k].c=b.data[j].c;
        c.data[k].d=b.data[j].d;
        k++;j++;
      }
     c.nums=k;
     }
      return true;                            //成功时返回 true
    }
```

5.4.2 稀疏矩阵的十字链表表示

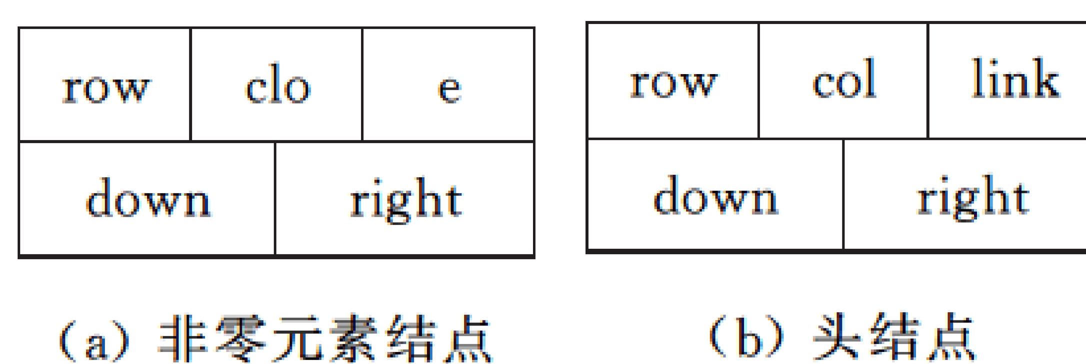
利用三元组表表示稀疏矩阵时,若矩阵的运算使非零元的个数发生改变,则必须对三元组表进行插入、删除,也就是必须移动表中的元素,由于三元组表是顺序存储,所以这些运算将花费大量的时间。十字链表作为一种链式存储结构可以克服上述缺点。

十字链表为稀疏矩阵的每一行设置一个单独链表,同时也为每一列设置一个单独链表。这样,稀疏矩阵的每一个非零元素就同时包含在两个链表中,即所在行的行链表中和所在列的列链表中。这就大大减少了链表的长度,方便了算法中行方向和列方向的搜索,因而大大降低了算法的时间复杂度。

对于一个 $m\times n$ 的稀疏矩阵,每个非零元素用一个结点表示,结点结构可设计成图 5.14(a)所示形式。其中,row、col、e 分别代表非零元素所在的行号、列号和相应的非零元素值;down 和 right 分别称为向下指针和向右指针,分别用来链接同列中和同行中的下一个非零元素结点。也就是说,稀疏矩阵中同一列的所有非零元素通过 down 指针链接成一个列链表,同一行中的所有非零元素通过 right 指针链接成一个行链表。对稀疏矩阵的每个非零元素来说,它既是某个行链表中的一个结点,同时又是某个列链表中的一个结点。每个非零元素就好像是在一个十字路口,由此称作十字链表。

十字链表中设置有行头结点、列头结点和链表头结点。它们采用和非零元素结点类似的结点结构,具体如图 5.14 所示。其中,行头结点和列头结点的 row、col 域值不置任何有意义的值;行头结点的 right 指针指向对应行链表的第一个结点,它的 down 指针为空;列头结点的 down 指针指向对应列链表的第一个结点,它的 right 指针为空。行头结点和列头结点必须顺序链接,这样,当需要逐行(列)搜索时,才能一行(列)搜索完后顺序搜索下一行(列)。行头结点和列头结点均用 link 指针完成顺序链接。对比行头结点和列头结点可以

看出,行头结点中未用 down 指针,列头结点中未用 right 指针,link 指针完成行或列头结点的顺序链接,row 域和 col 域未用。因此,行和列的头结点可以合用,即第 i 行和第 i 列头结点共用一个头结点。我们称这些合并后的头结点为行列头结点,行列头结点为矩阵行数 m 和列数 n 的最大值。



(a) 非零元素结点　　(b) 头结点

图 5.14　十字链表结点类型

十字链表头指针 mh 指向链表头结点,链表头结点的 row、col 域分别存放稀疏矩阵的行数 m 和列数 n,链表头结点的 link 指针指向行列头结点链表中的第一个行列头结点。由于矩阵运算中常常是一行(列)操作完后进行下一行(列)操作,所以十字链表中的所有单链表均链接成循环链表,这样就可方便地完成一行(列)操作后又回到该行(列)头结点,有 link 指针进入下一行列头结点,开始下一行(列)的操作。

下面是 3 行 4 列的稀疏矩阵。

$$B_{3\times4}=\begin{bmatrix}0&0&0&1\\0&2&0&0\\0&0&3&0\end{bmatrix}$$

在此稀疏矩阵中共有 3 个非零元素,分别用 4 个(3,4,3),(1,4,1),(2,2,2),(3,3,3)三元组表示。稀疏矩阵 B 对应的十字链表如图 5.15 所示。

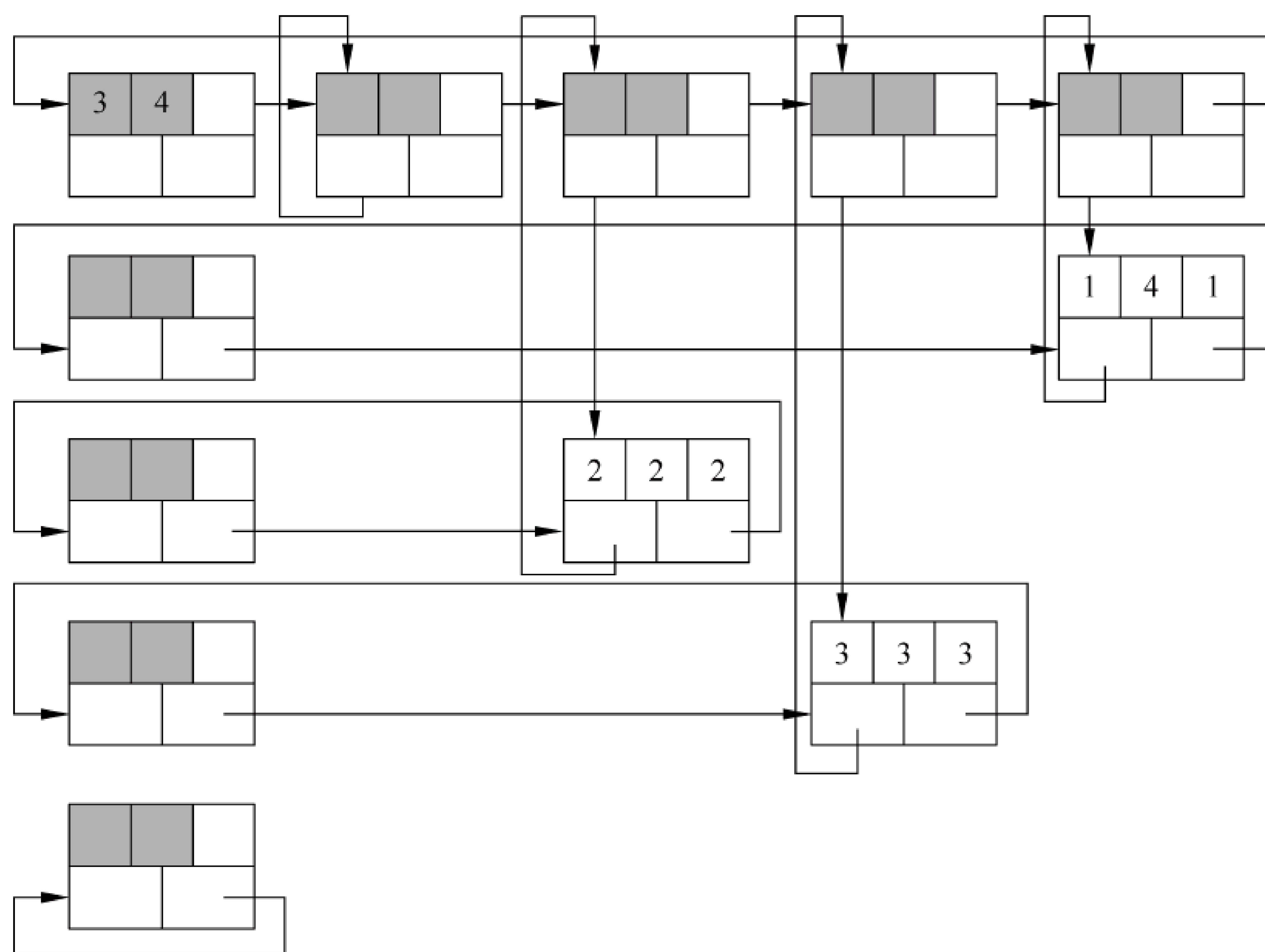


图 5.15　稀疏矩阵 B 对应的十字链表

为方便解释清楚，把每个行列头结点分别画成两个，而实际上，行头结点 h[i]($0 \leq i \leq 3$)与列头结点 h[i]是同一个结点，即 h[i]－＞down 域指向第 i 列的第一个结点，h[i]－＞right 域指向第 i 行的第一个结点。

十字链表结点结构和头结点的数据类型定义如下。

```
#define M <稀疏矩阵行数>
#define N <稀疏矩阵列数>
#define Max ((M)>(N)?(M):(N))                    //矩阵行列较大者
typedef struct mtxn
{   int row;                                     //行号
    int col;                                     //列号
    struct mtxn *right, *down;                   //向右和向下的指针
    union
    {   elemtype value;                          //非零元素值
        struct mtxn *link;                       //指向下一个头结点
    }tag;
}MatNode;                                        //十字链表类型定义
```

下面讨论十字链表的创建和输出运算算法。

1. 对一个二维矩阵创建其十字链表表示

先建立十字链表头结点的循环链表，然后以行序描述二维矩阵 A，将非零的元素插到十字链表中。插入操作如下。

step1：创建一个结点 *p。

step2：根据行号找到在行表中的插入位置并在行表中插入该结点。

step3：根据列号找到在列表中的插入位置并在列表中插入该结点。

对应的算法如下。

```
void  CreatMat(MatNode *&mh,elemtype a[M][N])
{   int i,j;
    MatNode *h[Max], *p, *q, *r;
    mh=(MatNode *)malloc(sizeof(MatNode));          //创建十字链表的头结点
    mh->row=M;mh->col=N;
    r=mh;                                           //r指向尾结点
    for(i=0;i<Max;i++)                              //采用尾插法创建头结点 h[0],
                                                      h[1],…循环链表
{   h[i]=(MatNode *)malloc(sizeof(MatNode));
    h[i]->down=h[i]->right=h[i];                    //将 down 和 right 方向置为循环的
    r->tag.link=h[i];                               //将 h[i]加到链表中
    r=h[i];
    }
    r->tag.link=mh;                                 //置为循环链表
    for(i=0;i<M;i++)                                //处理每一行
    { for(j=0;j<N;j++)                              //处理每一列
      {   if(a[i][j]!=0)                            //处理非零元素
              {p=(MatNode *)malloc(sizeof(MatNode)); //创建一个新结点
```

```
                        p->row=i;p->col=j;p->tag.value=a[i][j];
                        q=h[i];                              //查找在行表中的插入位置
                        while(q->right!=h[i]&&q->right->col<j)
                                q=q->right;
                        p->right=q->right;q->right=p;        //完成行表的插入
                        q=h[j];                              //查找在列表中的插入位置
                        while(q->down!=h[j]&&q->down->row<i)
                                q=q->down;
                        p->down=q->down;q->down=p;           //完成列表的插入
                    }
                }
            }
        }
```

对于建立十字链表的算法，如果非零元素是用户输入的，则时间复杂度为 O(Max×t)，其中 t 为非零元素个数，Max=max{M,N}，对非零元素输入的先后次序并无任何要求。若以行序为主输入三元组，则时间复杂度为 O(t)。上面的算法是从二维矩阵中获得非零元素的，它的时间复杂度为 O(M×N×Max)，因此它不是一个高效的算法。

2. 输出十字链表矩阵

以行序方式从头到尾扫描十字链表 h，依次输出元素值。

```
void  DispMat(MatNode *mh)
{   MatNode *p, *q;
    printf("行=%d 列=%d\n",mh->row,mh->col);
    p=mh->tag.link;
    while(p!=mh)
    {   q=p->right;
        while(p!=q)          //输出一行非零元素
        {   printf("%d\t%d\t%d\n",q->row,q->col,q->tag.value);
            q=q->right;
        }
        p=p->tag.link;
    }
}
```

【例 5.5】 设计一个用于存储双层集合的存储结构。双层集合是指集合中的每个元素本身也是一个集合(称为集合元素)，且由普通的元素构成。例如，s={{1,3},{1,7,8},{5,6}}。

解：采用类似于十字链表的思路，将每个集合元素设计成带头结点的单链表，再将这些集合元素头结点串起来构成一个单链表，设置 *h 作为集合头结点，如图 5.16 所示。

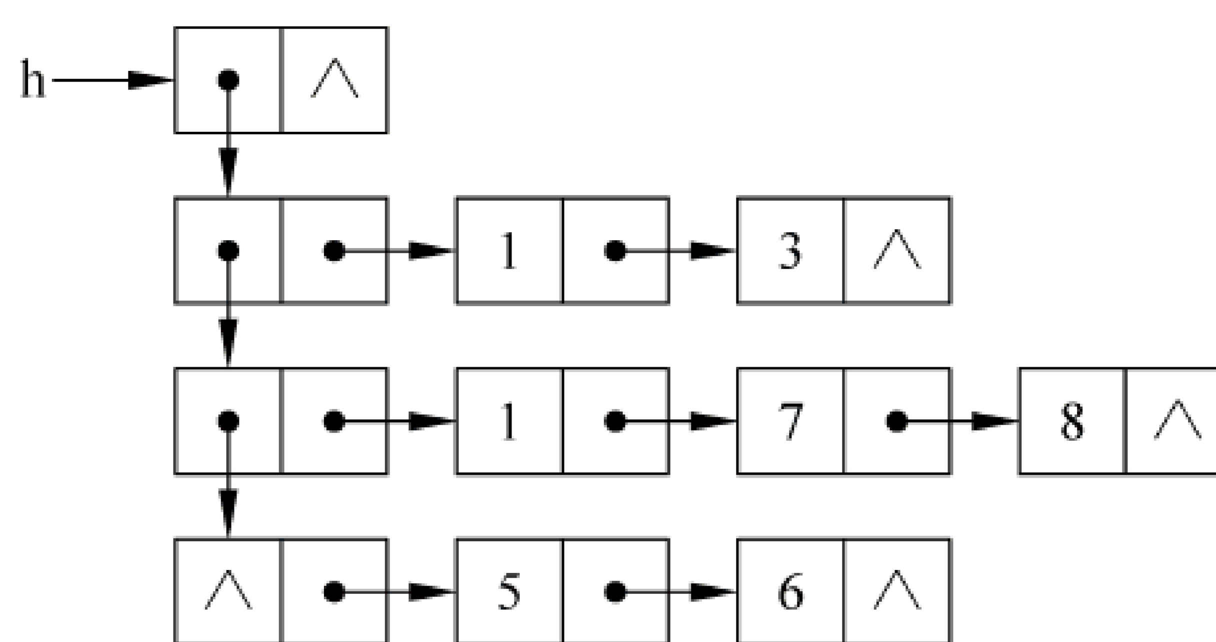


图 5.16 十字链表结点结构

数据结点的类型定义如下。

```
typedef struct dnode
{   elemtype data;
    struct dnode * next;
}DType;
```

集合元素头结点的类型定义如下。

```
typedef struct hnode
{   DType * next;
    struct hnode * link;
}HType;
```

集合头结点的类型与集合元素头结点的类型相同。

5.5 广 义 表

广义表作为线性表的推广,广泛用于人工智能等领域的表处理语言——LISP 语言,它把广义表作为基本的数据结构,连同程序也表示一系列的广义表。

5.5.1 广义表的定义

视频讲解

广义表简称**列表**。一个广义表是由 n($n \geq 0$)个元素组成的一个有限序列。设广义表 GL 的一般表示与线性表相同:$GL=(a_1,a_2,\cdots,a_i,\cdots,a_n)$,则 a_i 为广义表的第 i 个元素。

其中,n 表示广义表的长度,即广义表中所含元素的个数。特别地,n=0 时的广义表称为空表。线性表中的每个元素都具有相同的数据类型,但现实中通常会出现复杂的情况,即元素类型各不相同。如果 a_i 是单个数据元素,则 a_i 是广义表 GL 的原子(一般用小写字符表示);如果 a_i 是一个广义表,则 a_i 是广义表 GL 的**子表**(一般用大写字符表示)。结合定义看,广义表中元素的逻辑关系似乎仍与线性表一样属于线性结构,但是广义表中各个元素的元素类型不一定都相同,有可能是原子,也有可能是子表,所以广义表可以看成是线性表的推广。

若用圆圈和方框分别表示子表和原子,并用线段把表和它的元素(元素结点应在其表结点的下方)连接起来,则可得到一个广义表的图形表示。图 5.17 为广义表 D 的图形表示。

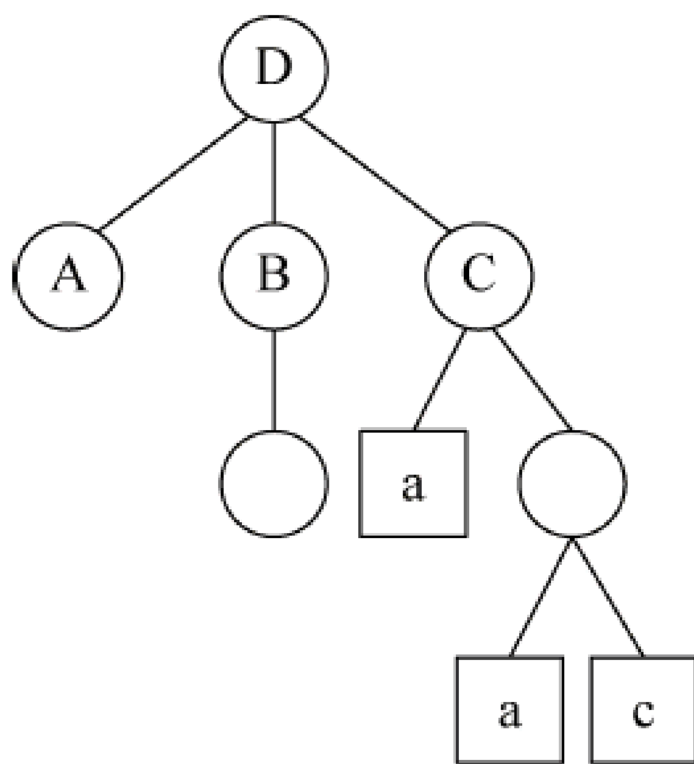


图 5.17 广义表 D 的图形表示

广义表具有如下重要的特性。

- 广义表中的数据元素有相对次序。
- 广义表的**长度**定义为最外层括号所包含的元素个数。
- 广义表**深度**为彻底展开广义表后所包含弧的重数。其中,原子的深度为 0,空表的深度为 1。
- 广义表可以共享。一个广义表可以被其他广义表共享,这种共享广义表称为**再入表**。
- 广义表可以是一个递归的表。一个广义表可以是自己的子表,这种广义表称为**递归**

表。递归表的深度是无穷值,长度是有限值。

- 任何一个非空广义表 GL 均可分解为**表头** Head(GL)=a_1 和**表尾** Tail(GL)=(a_2,…,a_n)两部分。

注意:空表 A=( ),因为没有元素,所以没有表头和表尾。

下面给出一些广义表的例子。

A=( )——A 是一个空表,其长度为 0,深度为 1。

B=(( ))——广义表 B 只有一个子表元素( ),B 的长度为 1,深度为 2。

C=(a,(a,c))——广义表 C 的长度为 2,两个元素分别是原子 a,子表(a,c),深度为 2。

D=(A,B,C)——D 有 3 个子表 A、B、C,D 的长度为 3,深度为 3。

E=(a,E)——E 是一个递归的表,E 的长度为 2,E 相当于一个无限的表(a,(a,(a,(…)))),深度为无穷大。

根据前面表述的表头、表尾可知,表的第一个元素 a_1 为广义表 GL 的**表头**。任何一个非空广义表的表头可能是原子,也可能是子表。其余部分(a_2,…,a_i,a_{i+1},…,a_n)为 GL 的**表尾**。广义表的表尾一定是子表。

A 是空表,无表头、表尾。

```
Head(B)=( ),Tail(B)=()
Head(C)=a,Tail(C)=((a,c))
Head(D)=A=( ),Tail(D)=(B,C)=((( )),(a,(a,c)))
Head(E)=a,Tail(E)=(E)
```

抽象数据类型广义表的定义如下。

```
ADT Glist
{   数据对象:
    D={ei|i=1,2,…,n;n≥0;ei∈AtomSet 或 ei∈GList,AtomSet 为某个数据对象}
    数据关系:
    R={<ei-1,ei>|ei-1,ei∈D,2≤i≤n}
    基本运算:
    CreateGL(&L):创建广义表.
        操作结果: 由括号表示法创建广义表.
    GLLength(L):求广义表长度.
        初始条件:广义表 L 已存在.
        操作结果:计算广义表 L 中元素的个数.
    GLDepth(L):求广义表深度.
        初始条件: 广义表 L 已存在.
        操作结果:计算广义表 L 最大括号层数.
    DispGL(l): 输出广义表 L.
        初始条件:广义表 L 已存在.
        操作结果:依次输出广义表 L 中的元素.
} ADT Glist
```



视频讲解

5.5.2 广义表的存储结构

广义表是一种递归的数据结构,由于其元素类型不像线性表一样统一,因此很难为每个

广义表分配固定大小的存储空间，所以其存储结构只能采用动态的链式存储结构。

广义表有两类结点：一类为圆圈结点，在这里对应子表；另一类为方形结点，在这里对应原子。为了使子表和原子两类结点既能在形式上保持一致，又能进行区别，可采用如图 5.18 所示的结构形式。

tag	first/data	link

图 5.18　广义表结点的结构形式

其中，tag 域为标志字段，用于区分两类结点。first/data 域由 tag 决定。若 tag=0，表示该结点为原子结点，则对应的结点的第二个域为 data，用于存放相应原子的数值信息；若 tag=1，表示该结点为子表结点，则第二个域为 first，存放相应子表第一个元素对应结点的地址。link 域存放与本元素同一层的下一元素所在结点的地址，当本元素是所在层的最后一个元素时，link 域为 NULL。

采用 C/C++语言描述结点的类型，可用如下定义。

```
typedef struct lnode                        //结点类型标识
{   int tag;
    union
    { elemtype data;                        //原子值
      struct lnode * first;                 //指向子表的第一个元素
     }val;
    struct lnode * link;                    //指向下一个元素
}GLNode;
```

5.5.1 节中列举的广义表的例子，它们的存储结构如图 5.19 所示。

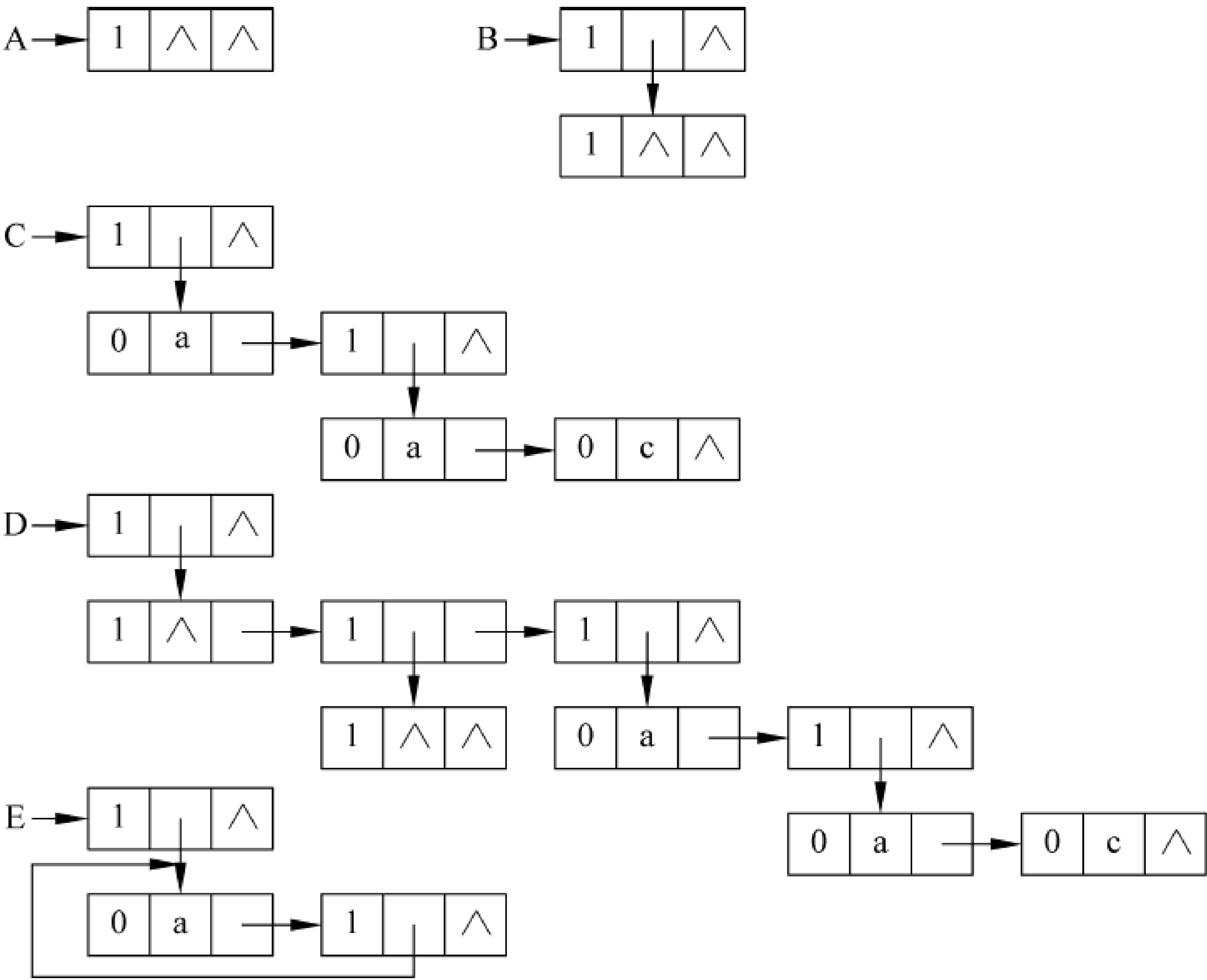


图 5.19　广义表的存储结构

5.5.3 广义表的运算



广义表的运算主要有求广义表的长度、求广义表的深度、输出广义表和建立广义表的链式存储结构等。由于广义表是一种递归的数据结构,所以对广义表的运算一般采用递归的算法。为了方便起见,之后的描述均用“(#)”表示空表。

1. 求广义表的长度

在广义表中,同一级别的每个结点都是通过 link 域链接起来的,所以可把广义表看作是由 link 域链接起来的单链表。如图 5.20 所示,将广义表看成带头结点的单链表。

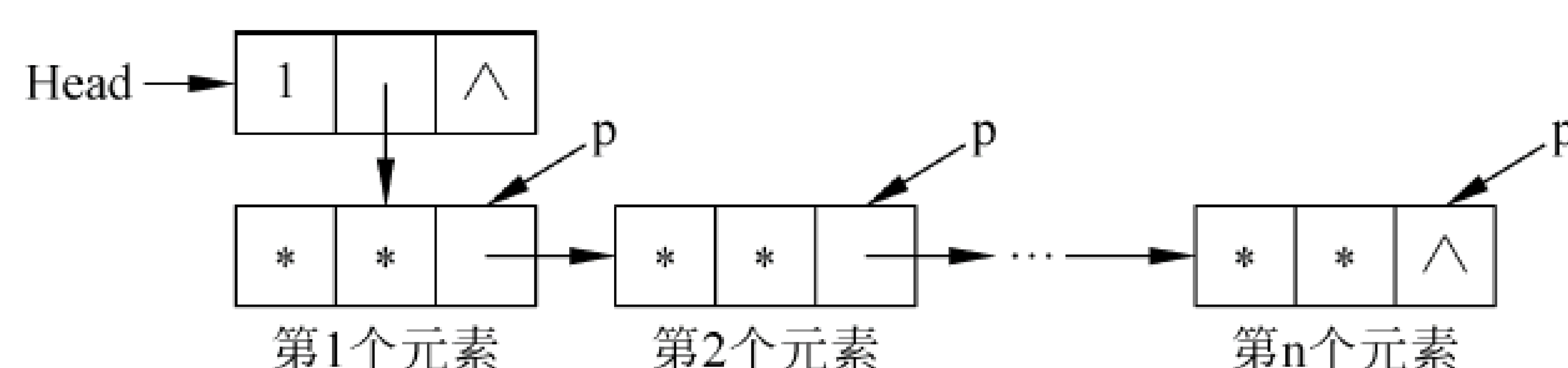


图 5.20 非空广义表计算长度分析

求广义表长度的非递归算法如下。

```
int  GLLength(GLNode *G)                        //求广义表 G 的长度
{  int n=0;
   GLNode *p=G->val.first;                      //G1 指向广义表的第一个元素
   while(p!=NULL)
   {  n++;                                      //累加元素个数
    p=p->link; }
   return n;
}
```

2. 求广义表的深度

对于广义表 G,其深度的递归定义是其元素的深度的最大值加 1。若 G 为原子,则其深度为 0。求广义表深度的递归模型如下。

```
f(G)=0                          //若 G 为原子
f(G)=1                          //若 G 为空表
f(G)=k+1                        //若 G 为非空广义表,则元素的深度的最大值为 k
```

假设广义表 D=(( ),(( )),(a,(a,c))),则其存储结构及计算深度的过程如图 5.21 所示。

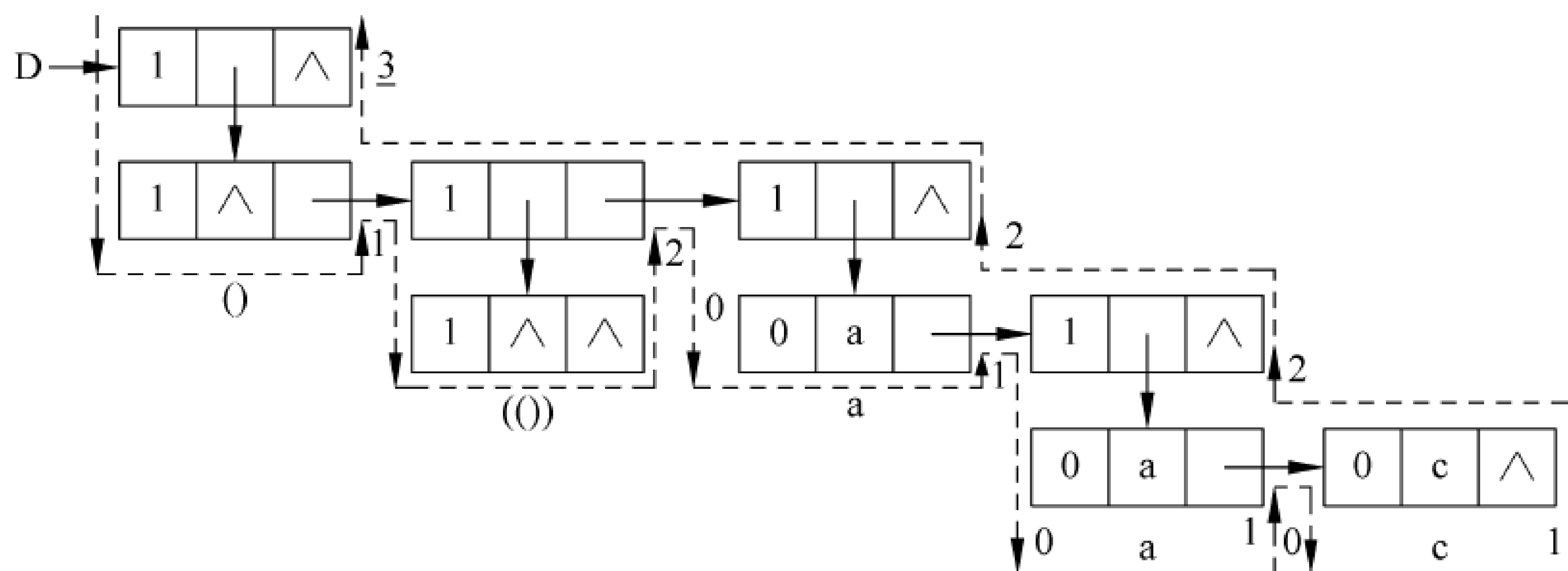


图 5.21 广义表 D 计算深度的过程

对于广义表 G,求其深度的算法如下。

```
int  GLDepth(GLNode *G)                    //求广义表 G 的长度
{
GLNode *G1;
int max=0,dep;
if(G->tag==0)  return 0;                   //为原子时返回 0
   G1=G->val.first;                        //G1 指向第一个元素
if(G1==NULL)  return 1;                    //为空表时返回 1
  while(G1!=NULL)                          //遍历表中的每一个元素
  {  if(G1->tag==1)                        //元素为子表的情况
     {dep=GLDepth(G1);                     //递归调用求出子表的深度
      if(dep>max)                          //max 为同一层所求过的子表中深度的最大值
        max=dep;
      }
       G1=G1->link;                        //使 G1 指向下一个元素
}
 return (max+1);                           //返回表的深度
}
```

3. 输出广义表

对于非空广义表 G,G->val.first 指向表中第一个元素,G->link 指向其兄弟。广义表的递归算法通常是先递归处理 G 中的元素,再递归处理 G 的兄弟。

输出广义表 G 的过程 f(G)为:若 G 不为 NULL,先输出 G 的元素,当有兄弟时,再输出兄弟。输出 G 的元素的过程是:如果 G->val.first 该元素为原子,则直接输出原子值,若为子表,则输出"(",接下来对子表进一步判断,若为空表,则输出"#",若为非空子表,则递归调用 f(G->val.first)输出子表,待子表输出完毕,再输出")"。输出 G 的兄弟的过程是:输出",",再递归调用 f(G->link)输出兄弟。

输出一个广义表的算法如下。

```
void  DispGL(GLNode *G)                    //输出广义表 G
{ if(G!=NULL)                              //表不为空判断
 {                                         //先输出 G 的元素
  if(G->tag==0)                            //G 的元素为原子时
      printf("%c",G->val.data);            //输出原子值
  else                                     //G 的元素为子表时
  {  printf("(");                          //输出"("
     if(G->val.first==NULL)                //为空表时
        printf("#");
     else                                  //为非空子表
        DispGL(G->val.first);              //递归输出子表
        printf(")");                       //输出")"
  }
  if(G->link!=NULL)
  {
    printf(",");
```

```
        DispGL(G->link);                    //递归输出 G 的兄弟
      }
    }
  }
```

4. 建立广义表的链式存储结构

假定广义表中原子的元素类型为 char,则每个原子的值被限定为单个小写英文字母。假定广义表是一个正确的表达式,其格式为:元素之间用一个逗号分隔,表元素的起止符号分别为左、右圆括号,空表在其圆括号内不包含任何字符。例如,“(a,(b,c,d),(#))”就是一个符合上述规定的广义表。

建立广义表存储结构的算法同样是一个递归算法。该算法使用一个具有广义表格式的字符串参数 s,返回由它生成的广义表存储结构的头结点指针 G。

在算法的执行过程中,需要从头到尾扫描 s 的每一个字符。当碰到“(”时,表明它是一个表或子表的开始,则应建立一个由 G 指向的表或子表结点,并用它的 first 域作为子表的表头指针进行递归调用,建立子表的存储结构;当碰到一个英文字母时,表明它是一个原子,则应建立一个由 h 指向的原子结点;当遇到“)”字符时,表明前面的子表已处理完毕,则将 G 置为空;当遇到“#”字符时,表明前面的子表是空表,即 G->val.first 置为空。

当建立了一个由 h 指向的结点后,若接着遇到“,”时,表明该结点存在兄弟结点,需要建立当前结点(即由 G 指向的结点)的兄弟结点;当碰到其他字符时,表明当前结点没有兄弟了,即当前结点的 link 域置为空。

根据以上分析,对应生成的广义表的链表存储结构的算法如下。

```
GLNode *  CreateGL(char * &s)               //返回由括号表示法表示 s 的广义表链表存储结构
{
 GLNode * G;
 char ch= * s++;                            //取一个字符
 if(ch!='\0')                               //串未结束
 {  G=(GLNode * )malloc(sizeof(GLNode));    //创建一个新结点
  if(ch=='(')                               //当前字符为左括号时
  {
    G->taG=1;                               //新结点作为表头结点
    G->val.first=CreateGL(s);
  }                                         //递归构造子表并链接到表头结点
  else if(ch==')')
    G=NULL;                                 //遇到")"字符,G 置为空
  else if(ch=='#')                          //遇到"#"字符,表示空表
    G=NULL;
  else                                      //为原子字符
  {  G->tag=0;                              //新结点作为原子结点
     G->val.data=ch;
  }
 }
 else                                       //串结束,G 置为空
```

```
      G=NULL;
    ch= * s++;                              //取下一个字符
    if(G!=NULL)                              //串未结束,继续构造兄弟结点
      if(ch==',')                            //当前字符为","
         G->link=CreateGL(s);                //递归构造兄弟结点
      else                                   //没有兄弟了,将兄弟指针置为 NULL
         G->link=NULL;
    return G;                                //返回广义表 G
  }
```

该算法需要扫描输入广义表中的所有字符,并且处理每个字符都是简单的比较或赋值操作,其时间复杂度为 O(1),整个算法的时间复杂度为 O(n),n 表示广义表中所有字符的个数。这个算法中既包含子表的递归调用,也包含兄弟的递归调用,所以递归调用的最大深度不会超过生成的广义表中所有结点的个数,因而其空间复杂度为 O(n)。

【例 5.6】 设计一个算法,求给定的广义表 G 中的原子个数。

分析:设 f(G)表示广义表 G 中的原子个数,根据广义表的递归特性得到递归模型如下。

```
f(G)=0                                当 G=NULL
f(G)=1+f(G->link)                     当 G 为原子结点时
f(G)=f(G->val.first)+f(G->link)      当 G 为表或子表结点时
```

对应的递归算法如下。

```
int  atomnum(GLNode * G)                   //求广义表 G 中的原子个数
{  if(G!=NULL)
     { if(G->tag==0)
          return 1+atomnum(G->link);
       else
          return atomnum(G->val.first)+atomnum(G->link);
     }
else
     return 0;
}
```

5.6 综合案例

5.6.1 大整数相乘

1. 问题描述

某些应用(尤其是当代密码技术)需要对超过 100 位的十进制整数进行乘法运算。整数过大超过计算机字长,就需要特别处理。两个 n 位整数相乘,需 n^2 次乘法。适当地减少乘法次数,增加加法次数可提高算法效率。

2. 解题思路

每一个整数都可以按照权值展开，例如：

计算 23×14：

$23 = 2\times10^1 + 3\times10^0, 14 = 1\times10^1 + 4\times10^0,$

$23\times14 = (2\times10^1+3\times10^0) \times (1\times10^1+4\times10^0)$

$= (2\times1)10^2 + (3\times1+2\times4)10^1 + (3\times4)10^0$　　4 次乘法

存储 2×1、3×4(2 次乘法)的结果：　　共 3 次乘法

$3\times1+2\times4 = (2+3) \times (1+4) - (2\times1) - (3\times4)$　　增加 1 次乘法

两个 2 位数相乘(十进制)

$$c=a\times b=c_2\,10^2+c_1\,10^1+c_0$$

$$a=a_1\cdot a_0, b=b_1\cdot b_0$$

例如：

$a=23, a_0=3, a_1=2;$

$c_2=a_1\times b_1, c_0=a_0\times b_0$

$c_1=(a_1+a_0)\times(b_1+b_0)-(c_2+c_0)$

因此，两个 n 位数相乘(设 n 为正偶数，a 分别为 a_1a_0 两部分，b 也如此)

$c=a\times b=c_2 10^n+c_1 10^{n/2}+c_0$

$a=a_1a_0=a_1 10^{n/2}+a_0, b=b_1b_0=b_1 10^{n/2}+b_0$

$c_2=a_1\times b_1, c_0=a_0\times b_0$

$c_1=(a_1+a_0)\times(b_1+b_0)-(c_2+c_0)$

若 n/2 也是偶数，用同样的方法计算 c_2, c_1, c_0。因此，若 $n=2^k$，得到计算 n 位数积的递归算法，当 n=1 或足够小的时候就停止。因此，这种分治递归法的算法实现大致如下。

```
int mult(x, y, n) {            //n 位正整数 x,y 计算乘积
   if(x == 0 || y == 0)
     return 0;
   if(n == 1)
     return x * y;
   else {
     int xl = x / (int)pow(10., (int)n/2);
     int xr = x - xl * (int)pow(10., n/2);
     int yl = y / (int)pow(10., (int)n/2);
     int yr = y - yl * (int)pow(10., n/2);
     int xlyl = mult(xl, yl, n/2);
     int xryr = mult(xr, yr, n/2);
     int xlxryryl = mult(xl-xr, yr-yl, n/2) + xlyl + xryr;
     return  (xlyl * (int)pow(10.,n)) +(xlxryryl * (int)pow(10.,n/2)) + xryr;
   }
}
```

当然，分治递归算法效率虽高，但是理解起来有难度。采用传统的数组存储方式也可以计算乘积，但效率较低，使用 3 个数组分别存放要相乘的两个数和乘积结果。例如，1234×123。

```
    1234
×    123
————————
    3702
   2468
  1234
————————
  151782
```

把乘数和被乘数、乘积结果都转化成数字型字符串处理，但字符串没有数学计算的功能，所以还要把字符串分拆成一个一个的数字，这样既能计算，又能处理很长的数据。

```
#include <iostream>
#include <memory>
using namespace std;
int * multi(int * num1, int s1, int * num2, int s2, int * num) {
  int s = s1 + s2;
  int * num = new int[size];
  int i = 0;
  memset(num, 0, sizeof(int) * size);
  for(i = 0; i < s2; i++) {
    int k = i;
    for(int j = 0; j < s1; j++)
      num[k++] = num2[i] * num1[j];
  }
  for(i = 0; i < s; i++) {
    if( num[i] >= 10) {
      num[i + 1] += num[i] / 10;
      num[i] %= 10;
    }
  }
  return num;
}
```

5.6.2 荷兰国旗问题

众所周知，荷兰国旗由红色、白色和蓝色 3 种颜色构成。现有红、白、蓝 3 个不同颜色的小球乱序排列在一起，请重新排列这些小球，使得红、白、蓝三色的同颜色的球在一起。该问题可作为一个数组排序问题处理。若红、白、蓝色球分别对应数字 0、1、2，且红、白、蓝色小球数量并不一定相同。遍历数组，统计红、白、蓝三色球(0,1,2)的个数，根据红、白、蓝三色球(0,1,2)的个数重排数组。

```
void sortcolor(int a[ ], int n) {
  if(n <= 1)
    return;
  int red = 0, white = 0, blue = 0;
  for(int i = 0; i < n; i++) {                    //分别统计红、白、蓝球的个数
    if(a[i] == 0) red++;
    elseif(a[i] == 1) white++;
    else blue++;
  }
  for(i = 0; i < n; i++) {                        //重新布局
    if(red > 0) {
      a[i] = 0;
      red--;
    } elseif(white > 0) {
      a[i] = 1;
      white--;
    } else {
      a[i] = 2;
        }
  }
}
```

本 章 小 结

本章主要介绍了数组和广义表的基本知识，主要学习要点如下。

- 理解数组的定义，特别是二维数组元素间的逻辑关系，实现二维数组和线性表的联系。
- 掌握二维数组按行/按列存储方法，并会计算二维数组中元素的内存地址。
- 理解矩阵压缩存储的目的和法则，特别是特殊矩阵压缩存储的方法和过程实现。
- 理解稀疏矩阵的定义及压缩方法，实现三元组表的类型定义及系数矩阵的转置运算。
- 了解稀疏矩阵十字链表的存储方式。
- 理解广义表的定义及相关概念。
- 掌握广义表的存储方式、类型定义及基本操作(求长度、求深度、取表头、取表尾)。

第3篇　树形结构篇

第6章　树和二叉树

前面介绍了几种常用的线性结构，本章讨论树形结构。树形结构属于非线性结构，常用的树形结构有树和二叉树。树结构在客观世界中广泛存在，如人类社会的族谱和各种社会组织机构都可以用树形象表示。树在计算机科学中有非常广泛的应用。例如，现代计算机操作系统均采用树形结构组织文件和文件夹；又如，在编译程序中，可用树表示源程序的语法结构。又如，在数据库系统中，树形结构也是信息的重要组织形式之一。本章介绍树的基本知识和应用。

6.1　树

6.1.1　树的定义

树是一种最典型的树形结构。**树**(**Tree**)是由 n 个结点组成的有限集合。如果 $n=0$，它就是一棵空树，这是树的特例；如果 $n>0$，有且仅有一个特定的称为**根**的结点，其余结点可分为 $m(m\geqslant 0)$个互不相交的有限子集 $T_1,T_2,\cdots,T_m$，其中每一个子集合本身又是一棵符合定义的树，通常称这样的树为根结点的**子树**。

例如，树 T={A,B,C,D,E,F,G,H,I,J,K,L,M}，用树形结构表示 T 如图 6.1 所示。

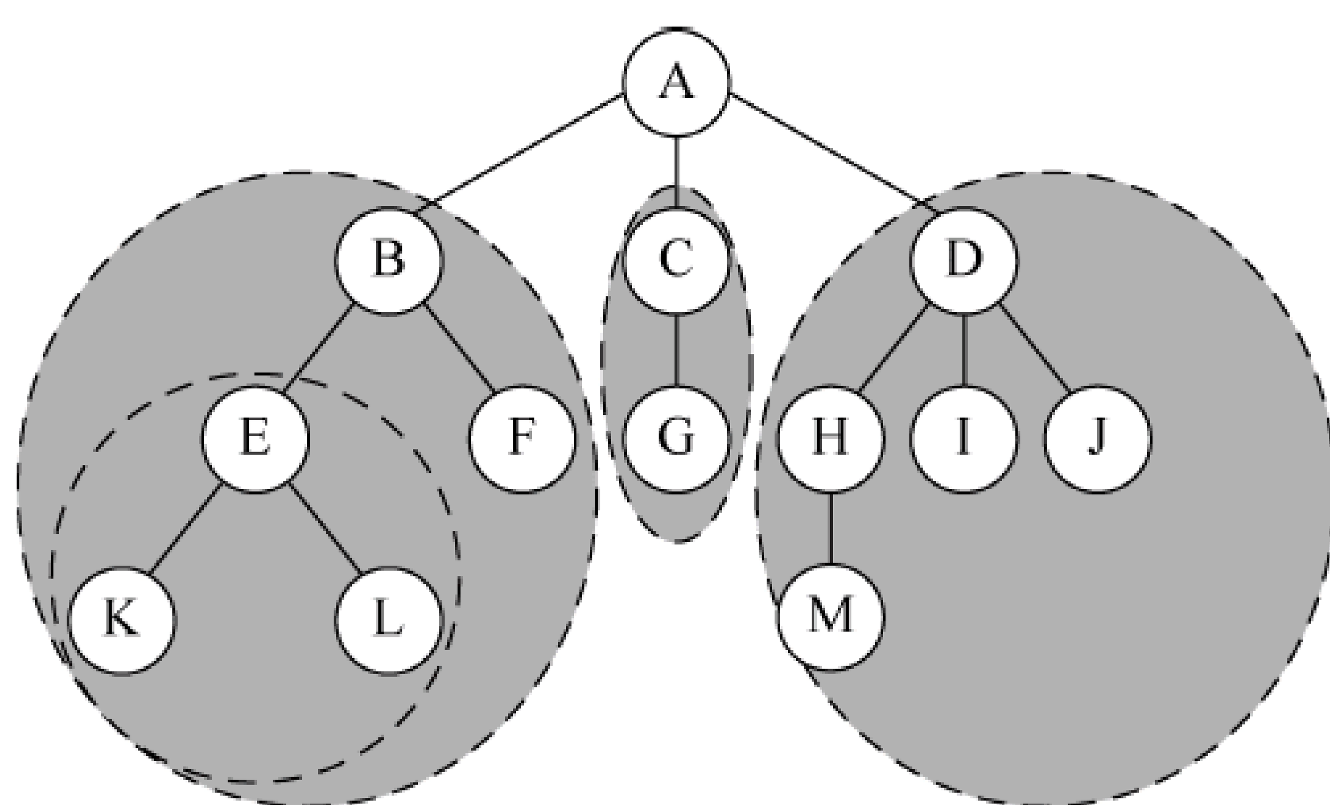


图 6.1　用树形结构表示 T

其中，A 是根结点，其余结点可以划分为 3 个互不相交的子集。

$T_1=\{B,E,F,K,L\}$，$T_2=\{C,G\}$，$T_3=\{D,H,I,J,M\}$，这些集合中的每一个集合本身又是一棵树，它们是 A 的子树。例如，子树 $T_{11}=\{E,K,L\}$，$T_{12}=\{F\}$，T_{11}、T_{12}是 B 的子树，B 是根，其余结点又可划分为 2 个互不相交的集合。

树的定义是递归的，因为在树的定义中又用到了树的定义，即一棵树由若干棵互不相交

的子树构成,而子树又由更小的若干棵子树构成。树是一种非线性数据结构,具有以下特点:它的每一个结点可以有零个或多个后继结点,但有且只有一个前驱结点(根结点除外);这些数据结点按分支关系组织起来,清晰地反映了数据元素之间的层次关系。可以看成数据元素之间存在的关系是一对多的关系。

6.1.2 树的术语



视频讲解

下面给出树形结构中的常用术语。

1. 结点的度

结点的度是指结点所拥有的子树个数(或者结点向下产生的分支个数)。

2. 树的度

树的度是指树中结点度的最大值。通常将度为 m 的树称为 m 次树(或者 m 元树)。

3. 分支结点

分支结点是指向下产生分支的结点(或者度>0 的结点)。

4. 终端结点

终端结点又称叶子结点,或向下不产生分支的结点,即度等于 0 的结点。

5. 结点的层次

若树中根结点的层次为 1,则根结点子树的根为第 2 层,以此类推。

6. 树的深度

树的深度是指树中所有结点层次的最大值。

7. 孩子、双亲

结点子树的根称为这个结点的**孩子**,而这个结点又被称为孩子的**双亲**。

8. 兄弟

同一个双亲的孩子之间互为**兄弟**。

9. 有序树和无序树

若树中各结点的子树是按照一定的次序从左向右排列的,且相对次序不能随意变换,则该树为**有序树**,否则为**无序树**。本文默认为有序树。

10. 森林

n(n>0)个互不相交的树的集合称为**森林**。森林的概念与树的概念很相近,如果删去一棵树的树根,留下的子树就构成了一个森林。当删去的是一棵有序树的树根时,留下的子树也是有序的,这些树组成一个树表。在这种情况下,通常这些树组成的森林称为有序森林或果园。

注意:

- 非空树中有且只有一个根结点,于是可以通过根结点是否存在判断树是否为空。
- “度为 0 的树”与“空树”不同,只有一个根结点的树度为 0,但并非空树。
- 理解“树的度”与“结点的度”之间的关系。例如,“如果树的度为 3,则树中结点度的最大值为 3,并非所有结点的度为 3”。同时,还可根据“结点的度”将树中的结点最多分为 4 类:度等于 0、度等于 1、度等于 2、度等于 3 的结点。
- 本教材中默认为有序树。

6.1.3 树的基本性质

性质 1 ：树的结点总数等于结点的度数之和加上 1。

如图 6.1 所示，树的结点总数 $n=13$，结点的度数之和 $e=12$，显然 $n=e+1$。此外，树的性质也可以描述为"树的结点总数等于分支总数加上 1"。显然，树中分支的个数和结点的度数刚好对应。例如，A 结点的度为 3，即 A 结点向下产生 3 个分支；B 结点的度为 2，即 B 结点向下产生 2 个分支；而 K 结点的度为 0，则 K 向下产生的分支个数为 0。

【例 6.1】 一棵度为 3 的树，其中度为 1、2、3 的结点个数分别是 2 个、1 个、2 个；计算叶子结点的个数？

分析：之前介绍过树的定义及相关术语可以帮助理解题目，但只有树的性质是一个与结点个数有关的等式。显然，该题目要利用树的性质进行求解。

解：假设该 3 次树中度为 0,1,2,3 的结点个数分别为 n_0 个，n_1 个，n_2 个，n_3 个，结点总数为 n 个，分支总数为 e 个，由树的性质可知 $\mathbf{n=e+1}$。

因为 $n=n_0+n_1+n_2+n_3=n_0+2+1+2=n_0+5$

又因为 $e=0*n_0+1*n_1+2*n_2+3*n_3=1*2+2*1+3*2=10$

而 $n_0+5=10+1$　所以 $n_0=6$

【例 6.2】 若给了 n 个结点（$n>0$），要构造一棵度为 2 的树，问所构造树的高度最大值是多少？最小值是多少？

分析：构造一棵度为 2 的二次树，若按照树形结构从上向下逐层构建，当每层上分布的结点尽可能少时，该树的高度就会越大；相反，若每层上分布的结点尽可能多时，该树的高度就会越小，但注意第一层上的根结点有且只能有一个。于是得出如下两种形态的树形结构，如图 6.2 所示。

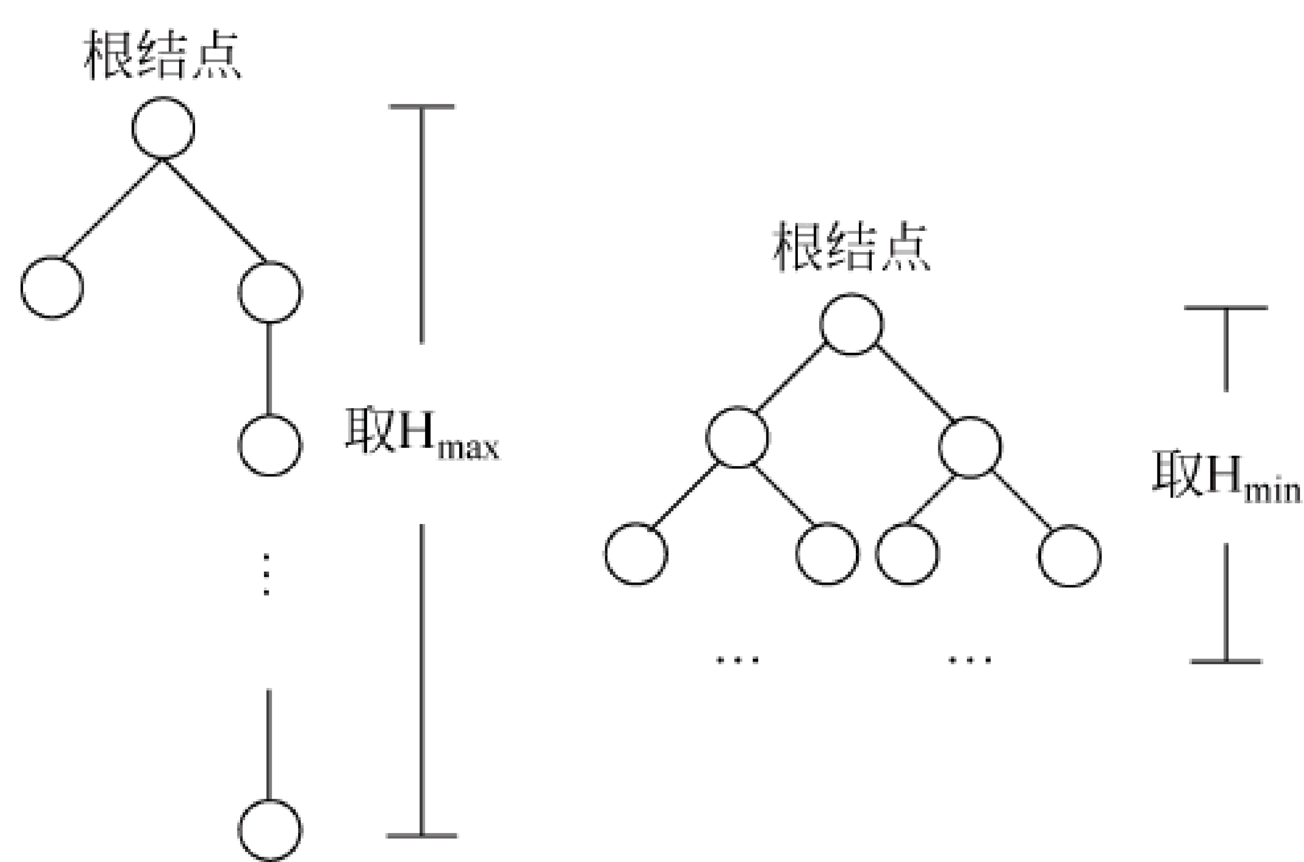


图 6.2　最大高度和最小高度的树

观察上面的树形结构可知 $H_{max}=n-1$ 容易计算，计算 H_{min} 时，设此时树共有 h 层，前 $h-1$ 层：从上向下，每层上结点个数分别为 $1,2^1,2^2,2^3,\cdots,2^{h-2}$ 个，最后一层的结点个数最少，为 1 个，最多为 2^{h-1} 个，于是得到 h 与结点总数 n 的如下关系。

前 $h-1$ 层结点数：$1+2^1+2^2+2^3+\cdots+2^{h-2}=2^{h-1}-1$ 个

于是 $2^{h-1}-1+1\leqslant n\leqslant 2^{h-1}-1+2^{h-1}$

所以 $2^{h-1}\leqslant n$ 且 $n\leqslant 2^h-1$

则 $h\leqslant \log_2 n+1$ 且 $\log_2(n+1)\leqslant h$

所以 $h=[\log_2 n+1]$

即 $H_{min}=[\log_2 n+1]$

6.1.4 树的抽象数据类型

树的抽象数据类型定义如下。

ADT Tree
{
数据对象:D={a_i|1≤n,n≥0,a_i 为 elemtype 类型}　　//elemtype 是自定义类型标识符
数据关系:R={<a_i, a_j>| a_i, a_j∈D,1≤i≤n,1≤j≤n,其中每个元素只有一个前驱结点(除根结点),可以有零个或多个后继结点,有且仅有一个元素没有前驱结点}
基本运算:
InitTree(&T):初始化树。
　操作结果:构造一棵空树 T。
DestroyTree(&T):销毁树。
　初始条件:树 T 已存在。
　操作结果:释放树 T 占用的存储空间。
Parent(T,p):计算 p 所指结点的双亲结点。
　初始条件:树 T 已存在。
　操作结果:返回 p 所指结点的双亲结点信息。
Sons(T,p):计算 p 所指结点的子孙结点。
　初始条件: 树 T 已存在。
　操作结果: 返回 p 所指结点的子孙结点信息。
TreeDepth(T):计算树 T 的深度。
　初始条件: 树 T 已存在。
　操作结果:计算并返回树 T 的深度。
TraverseTree(T):树的遍历。
　初始条件: 树 T 已存在。
　操作结果:按照某种方式,对树 T 中的每个结点访问一次。
}ADT Tree

6.2 二　叉　树



视频讲解

6.2.1 二叉树的定义

二叉树也称二分树,它是有限的结点集合,这个集合或者是空,或者是由一个根结点和两棵互不相交的称为左子树和右子树的二叉树组成。

抽象数据类型二叉树的定义和抽象类型树的定义相似,这里不再介绍。显然,和树的定义一样,二叉树的定义也是一个递归定义。二叉树的结构简单,存储效率高,其运算算法也相对简单,而任何 m 次树都可以转化为二叉树结构。因此,二叉树具有很重要的地位。

二叉树和度为 2 的树(二次树)是不同的,其差别表现在：对于非空树,

- 度为 2 的树中至少存在一个结点的度为 2,而二叉树没有这种要求。
- 度为 2 的树不区分左、右子树,而二叉树是严格区分左、右子树的。

【例 6.3】 由 3 个结点最多可构造多少种不同形态的二叉树?

解：答案为图 6.3 中的 5 种形态,强调了二叉树对子树分左、右的问题。

同样，由 3 个结点最多可构造多少种不同形态的树？

答案为图 6.4 中的 2 种形态。树对子树强调有序性，不需要分左、右。

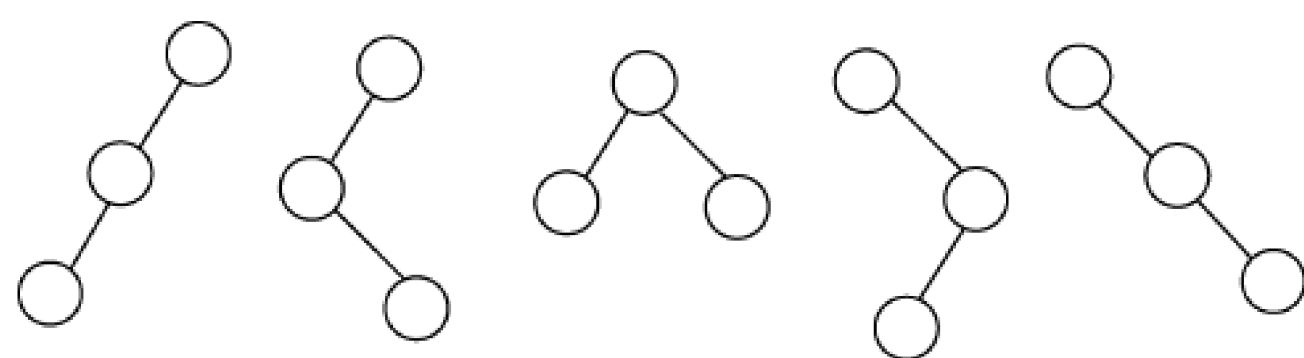


图 6.3　5 种不同形态的二叉树

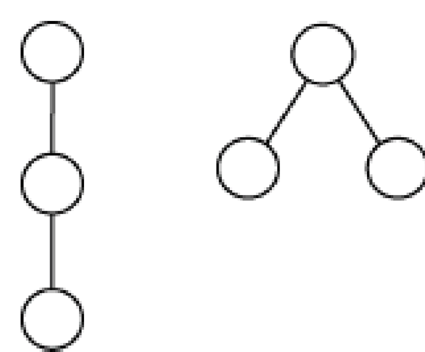
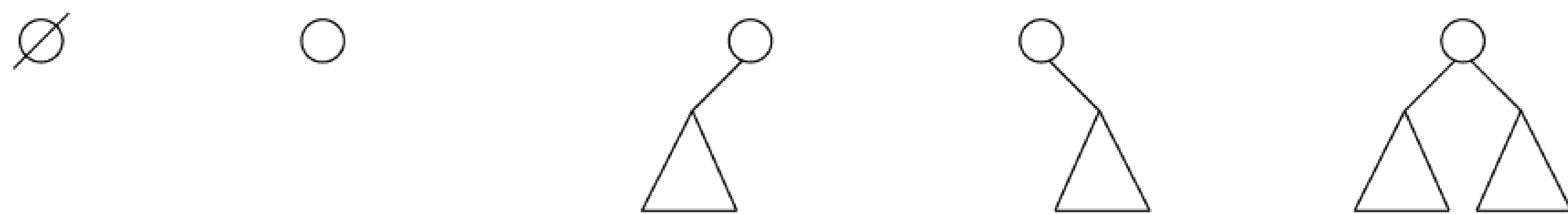


图 6.4　两种不同形态的树

一般二叉树有 5 种基本形态，如图 6.5 所示，任何复杂的二叉树都是这 5 种基本形态的复合。其中，图 6.5(a)是空二叉树，图 6.5(b)是只有一个根结点的二叉树，图 6.5(c)是整体右子树为空的二叉树，图 6.5(d)是整体左子树为空的二叉树，图 6.5(e)是左、右子树都不为空的二叉树。

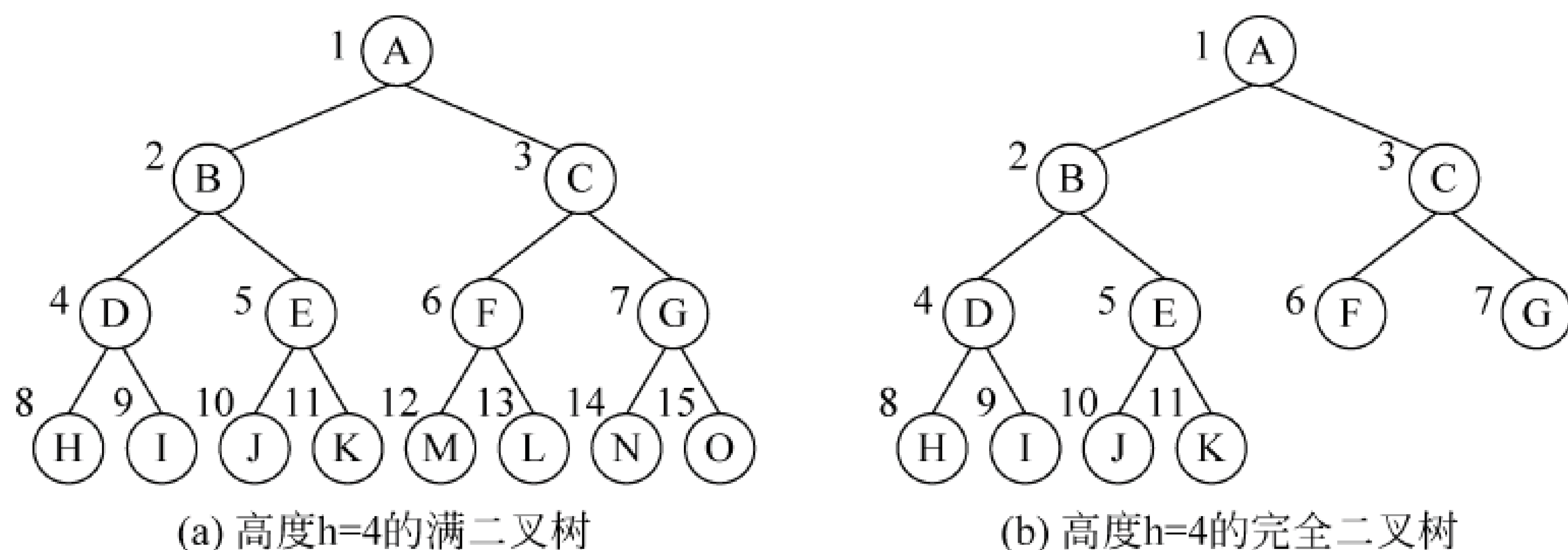


(a) 空树　(b) 只有一个根结点　(c) 整体只有左子树　(d) 整体只有右子树　(e)左、右子树均有

图 6.5　二叉树的 5 种基本形态

二叉树的表示法与树的表示法一样，有树形表示法、文氏图表示法、凹入表示法和括号表示法等。

在一棵二叉树中，如果所有分支结点都有左孩子结点和右孩子结点，并且叶子结点都集中在最下一层，这样的二叉树称为**满二叉树**。图 6.6(a)所示就是一棵满二叉树。可以对满二叉树的结点进行程序编号，约定编号从树根为 1 开始，按照层数从小到大、从左向右的次序进行，图 6.6(a)中每个结点外边的数字为该结点的编号。也可以从结点个数和树高度之间的关系定义满二叉树，即一棵高度为 h 具有 2^h-1 个结点的二叉树称为**满二叉树**。



(a) 高度h=4的满二叉树　(b) 高度h=4的完全二叉树

图 6.6　特殊的二叉树

满二叉树的特点如下。

- 叶子结点都在最下一层。
- 只有度为 0 和度为 2 的结点。
- 同高度的二叉树中，结点个数达到最大值。

若二叉树中最多只有最下面两层的结点数的度数小于 2，并且最下面一层的叶子结点数都依次排列在该层最左边的位置上，则这样的二叉树称为完全二叉树，如图 6.6(b)所示

为一棵完全二叉树。同样,可以对完全二叉树中的每个结点进行层序编号,编号的方法与满二叉树相同,图 6.6(b)中每个结点外面的数字为该结点的编号。

不难看出,满二叉树是完全二叉树的一种特殊情况,并且完全二叉树与等高度的满二叉树对应位置的结点有同一编号。图 6.6(b)的完全二叉树与等高度的满二叉树相比,它在最后一层的右边缺少 4 个结点。

完全二叉树的特点如下。

- 叶子结点只可能在层次最大的两层上出现。
- 对于最大层次中的叶子结点,都一次排列在该层的最左边的位置上。
- 如果有度为 1 的结点,只可能有一个,且该结点只有左孩子,而无右孩子。
- 按层序编号后,一旦出现某结点(其编号为 i)为叶子结点或只有左孩子,则编号大于 i 的结点均为叶子结点。
- 当结点的总数 n 为奇数时,度为 1 的结点个数 $n_1=0$;当结点的总数 n 为偶数时,$n_1=1$。

6.2.2 二叉树的性质

性质 1:非空二叉树上的叶子结点数等于双分支结点数加 1。

证明:设二叉树上叶子结点数为 n_0,单分支结点数为 n_1,双分支结点数为 n_2(如果没有特别指出,后面均采用这种设定),则总结点数 $n=n_0+n_1+n_2$。在一棵二叉树中,所有结点的分支数(即所有结点的度之和)应等于单分支结点数加上 2 倍双分支结点数,即总的分支数$=n_1+2n_2$。

由于二叉树中除根结点外,每个结点都有唯一的一个分支指向它,因此二叉树中有:总的分支数=总结点数-1。

由上述 3 个等式可得:$n_0+2n_2=n_0+n_1+n_2-1$

即 $n_0=n_2+1$

注意:在二叉树中计算结点个数时,常用的关系式有:

- 所有结点的度之和$=n-1$。
- 所有结点的度之和$=n_1+2n_2$。
- $n=n_0+n_1+n_2$。

性质 2:非空二叉树上第 i 层上至多有 2^{i-1} 个结点($i\geqslant 1$)。

由数学归纳法可知,二叉树第 1 层上最多 1 个结点,第 2 层上最多 2 个结点,第三层上最多 4 个结点,以此类推,可得出性质 2。

性质 3:高度为 h 的二叉树至多有 2^h-1 个结点($h\geqslant 1$)。

由二叉树的性质 2 可推出,高度为 h 的二叉树,若每一层上的结点数均达到最多,则每层上的结点数相加可得总结点数,必定最多,即 $1+2+2^2+\cdots+2^{h-1}=2^h-1$。

性质 4:对完全二叉树中的编号为 i 的结点($1\leqslant i\leqslant n, n\geqslant 1$,n 为结点数),有:

- 若 $i\leqslant[n/2]$,即 $2i\leqslant n$,则编号为 i 的结点数为分支结点,否则为叶子结点。
- 若 n 为奇数,则每个分支结点都既有左孩子,又有右孩子(例如,图 6.6(b)的完全二叉树就是这种情况,其中 $n=11$,分支结点 1~5 都有左、右孩子结点);若 n 为偶数,则编号最大(编号为 n/2)的分支结点只有左孩子结点,没有右孩子结点,其余分支结点都有左、右孩子结点,即 n 为奇数时,$n_1=0$;n 为偶数时,$n_1=1$。
- 若编号为 i 的结点有左孩子结点,则左孩子结点编号为 2i;若编号为 i 的结点有右孩

子结点,则右孩子结点的编号为 $2i+1$。

- 除根结点外,若一个结点的编号为 i,则它的双亲结点的编号为[n/2]。也就是说,当 i 为偶数时,其双亲结点的编号为 n/2,它是双亲结点的左孩子;当 i 为奇数时,其双亲结点的编号为(i−1)/2,它是双亲结点的右孩子结点。

上述性质均可采用归纳法证明,读者可自行完成。

性质 5:具有 n 个(n>0)个结点的完全二叉树的高度为$\lceil \log_2(n+1) \rceil$或$\lfloor \log_2(n) \rfloor+1$。

由完全二叉树的定义和树的性质 3 可推出。

说明: 对于一棵完全二叉树,结点总数 n 可以确定其形态,n_1 只能是 0 或 1,当 n 为偶数时,$n_1=1$;当 n 为奇数时,$n_1=0$。

【例 6.4】 若一棵完全二叉树的结点总数为 n,则编号最大的分支结点的编号是多少?

解: 由二叉树的性质 1 可知 $n_0=n_2+1$,而二叉树结点度数之和$=2n_2+n_1$,

因此 $n=n_0+n_1+n_2=2n_2+n_1+1$,则 $n_2=\dfrac{n-n_1-1}{2}$

在完全二叉树中,n_1 只能为 0 或为 1。当 $n_1=0$ 时(此时 n 为奇数),二叉树只有度为 2 的结点和叶子结点,所以最大的分支编号就是 n_2,此时

$$n_2=\frac{n-1}{2}=[n/2]$$

当 $n_1=1$ 时(此时 n 为偶数),二叉树中只有一个度为 1 的结点(该结点是最后一个分支结点),此时最大的分支结点编号为 $n_2+1=[n/2]$。

归纳起来,编号最大的分支结点的编号是[n/2]。

【例 6.5】 一棵完全二叉树有 500 个结点,问该完全二叉树有多少个叶子结点?有多少个度为 1 的结点?有多少个度为 2 的结点?

解: 由性质 4 可知,n=500 是偶数,则 $n_1=1$;

由性质 1 可知 $n_0=n_2+1$,则 $n_2=n_0-1$;

由树的性质可知 $n=n_0+n_1+n_2$;

$500=n_0+1+n_0-1$

则 $n_0=250$,$n_2=249$

6.2.3 二叉树的抽象数据类型

二叉树的抽象数据类型定义如下。

```
ADT BiTree
{
数据对象:
数据对象:D={ai|1≤n,n≥0,ai 为 elemtype 类型}
数据关系:R={< ai, aj >| ai, aj ∈D,1≤i≤n,1≤j≤n,其中每个元素只有一个前驱结点(除根结点),可以有零个或最多 2 个被称为左、右孩子的后继结点,有且仅有一个元素没有前驱结点}
基本运算:
InitBiTree(&b):初始化二叉树。
    操作结果:构造一棵空二叉树 b。
DestroyBiTree(&b):销毁二叉树。
    初始条件:二叉树 b 已存在。
    操作结果:释放树 b 占用的存储空间。
```

Parent(b,s):求 s 所指结点的双亲结点。
  初始条件：二叉树 b 已存在。
  操作结果：返回二叉树中 s 所指结点的双亲结点。
Sons(b,s):求 b 所指结点的子孙结点。
  初始条件：二叉树 b 已存在。
  操作结果：返回二叉树中 s 所指结点的子孙结点。
TreeDepth(b):计算树 b 的深度。
  初始条件：二叉树 b 已存在。
  操作结果：计算并返回二叉树 b 的高度。
TraverseTree(b):二叉树的遍历。
  初始条件：二叉树 b 已存在。
  操作结果：按照某种方式对二叉树 b 中的每个结点访问一次。
} ADT BiTree

6.2.4 二叉树的存储结构



视频讲解

二叉树的存储结构主要有顺序存储和链式存储两种。

1. 二叉树的顺序存储结构

二叉树的顺序存储结构就是用一组地址连续的存储单元存放二叉树的数据元素。因此,必须确定好树中各数据元素的存放次序,使得数据元素相互位置能反映相互数据元素之间的逻辑关系。

二叉树中顺序存储结构中结点的存放次序依次是：对该树中的每个结点进行编号,其编号从小到大顺序就是结点存放在连续存储单元的先后次序。若把二叉树存储到一维数组中,则该编号就是下标值加 1(注意,C/C++语言中数组的起始下标为 0)。树中各结点的编号与等高度的完全二叉树中对应位置上结点的编号相同。其编号过程是：首先把树根结点的编号定为 1,然后按照从上到下、从左到右的顺序对每一结点进行编号。当某结点是编号为 i 的双亲结点的左孩子结点时,则它的编号应为 2i；当它是右孩子结点时,则它的编号应为 2i+1。

根据二叉树的性质 5,在二叉树的顺序存储中各结点之间的关系可通过编号(存储位置)确定。对于编号为 i 的结点(即第 i 个存储单元),其双亲结点的编号为 i/2；若存在左孩子结点,则其左孩子结点的编号为 2i；若存在右孩子结点,则其右孩子结点的编号为 2i+1。因此,访问每一个结点的双亲和左、右孩子结点(若有的话)都非常方便。

二叉树顺序存储结构类型定义如下。

```
typedef elemtype SqBTree[MaxSize];
```

其中,elemtype 为二叉树中结点值的类型,当二叉树中某结点为空结点或无效结点(不存在该编号的结点)时,对应位置的值用特殊值(如“#”)表示。

【例 6.6】 给出图 6.6(a)、(b)所示二叉树的顺序存储结果。

解：图 6.6(a)所示二叉树对应的顺序存储如下。

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

图 6.6(b)所示二叉树的顺序存储如下。

1	2	3	4	5	6	7	8	9	10	11
A	B	C	D	E	F	G	H	I	J	K

【例 6.7】 给出图 6.7 所示一般二叉树的顺序存储结果。

图 6.7 所示二叉树对应的顺序存储：先采用完全二叉树的编号方式，没有编号的结点在对应位置用"#"表示，此过程被称为虚化成等高度的完全二叉树（图 6.8）过程。

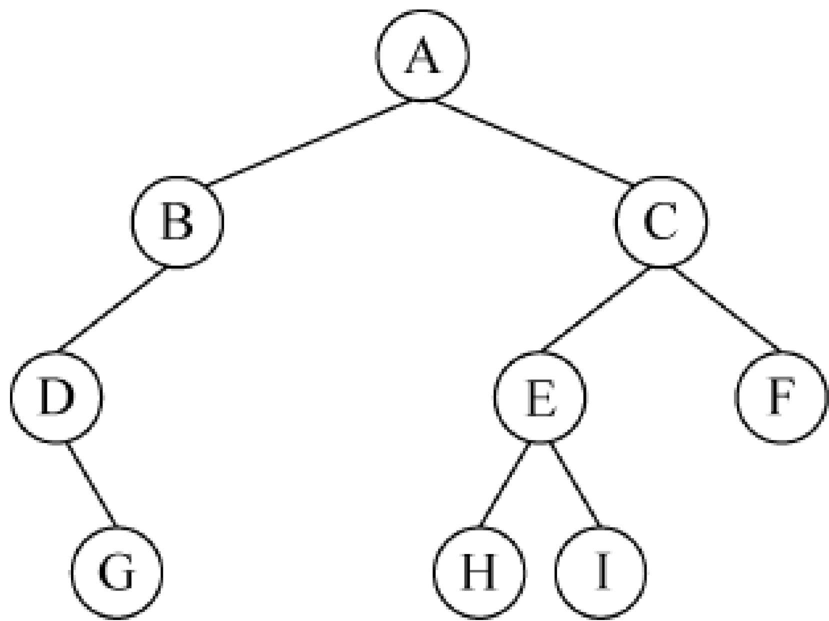


图 6.7　一般二叉树

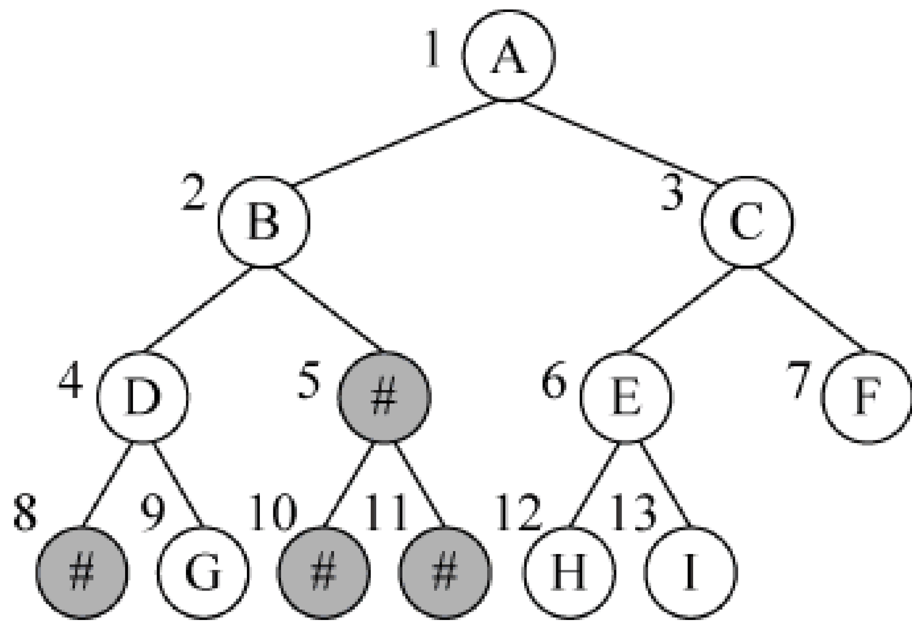


图 6.8　完全二叉树

例如，以下语句定义了图 6.7 所示二叉树的顺序存储结构（假定 elemtype 为 char 类型）。

```
SqBTree bt="ABCD#EF#G##HI";
```

1	2	3	4	5	6	7	8	9	10	11	12	13
A	B	C	D	#	E	F	#	G	#	#	H	I

其中，bt 数组的下标为 0 的位置不使用，赋特殊值，其他位置上的"#"字符表示空结点或无效结点。

于是，一般二叉树的顺序存储步骤大致分 3 步。

step1：一般的二叉树虚化成同高度的完全二叉树。

step2：对虚化后的完全二叉树从上向下，同层上从左到右进行结点的编号。

step3：将结点的编号当作结点在数组中的下标实现顺序存储。

对于完全二叉树，采用顺序存储方式是十分合适的，它能够充分利用存储空间。但对于一般的二叉树，特别是对于那些单分支结点比较多的二叉树来说，是很不合适的，因为可能只有少数存储单元被利用，尤其是对那些退化的二叉树（即每个分支结点都是单分支结点），空间浪费更是惊人。顺序存储结构固有的缺陷使得二叉树的插入、删除等运算十分不方便。因此，对于一般二叉树，通常采用链式存储方式。

2. 二叉树的链式存储结构

二叉树的链式存储结构是指用一个链表存储一棵二叉树，二叉树中的每一个结点用链表中的一个结点存储。二叉链表结点结构如图 6.9 所示。

lchild	data	rchild

图 6.9　二叉链表结点结构

其中，data 表示值域，用于存储结点的数据元素值，lchild 和 rchild 分别表示左孩子指针域和右孩子指针域，分别应用于存储左孩子结点和右孩子结点（即左、右子树的根结点）的存储位置。这种存储结构通常称为**二叉链表**。

对应的 C/C++语言的结点类型定义如下。

```
typedef struct BTNode //定义 BTNode 结点类型
{   elemtype data;                                //数据元素
    struct  BTNode  * lchild;                     //指向左孩子结点
    struct  BTNode  * rchild;                     //指向右孩子结点
}BiTree;                                          //定义 BiTree 二叉链表类型
```

本章后面的算法均用到二叉链表存储结构，为此可以将类型定义存储到头文件 btree.h 中。例如，图 6.10(a)的二叉树对应的二叉链存储结构如图 6.10(b)所示。

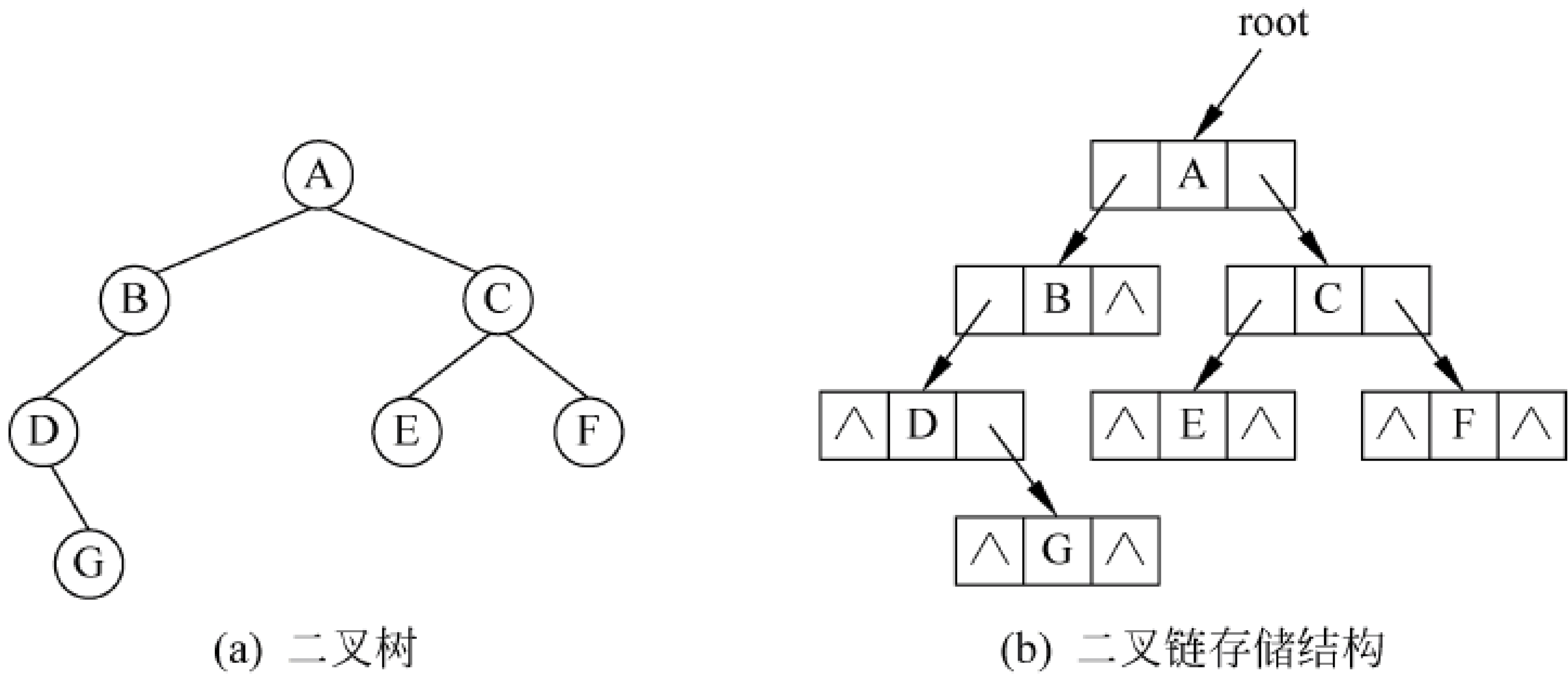


图 6.10　二叉树及其二叉链存储结构

注意：具有 n 个结点的二叉链表中，非空指针域为 n－1 个（和树的分支个数一一对应），空指针域为 n＋1 个。

6.3　二叉树的基本操作

6.3.1　中序遍历

中序遍历的过程是：

若二叉树非空，则

step1：中序遍历左子树。

step2：访问根结点。

step3：中序遍历右子树。

例如，图 6.9(a)的二叉树的中序序列为 DGBAECF。显然，在一棵二叉树的中序序列中，根结点值将其序列分为前、后两部分，前部分为左子树的中序序列，后部分为右子树的中序序列。

6.3.2　先序遍历

先序遍历二叉树的过程是：

若二叉树非空，则

step1：访问根结点。

step2：先序遍历左子树。

step3：先序遍历右子树。

例如，图 6.9(a)的二叉树的先序序列为：ABDGCEF。显然，在一棵二叉树的先序序列中，第一个元素即为根结点对应的结点的值。

6.3.3 后序遍历

后序遍历二叉树的过程是：

若二叉树非空，则

step1：后序遍历左子树。

step2：后序遍历右子树。

step3：访问根结点。

例如，图 6.9(a)的二叉树的后序序列为：GDBEFCA。显然，在一棵二叉树的后序序列中，最后一个元素即为根结点对应的结点值。

6.3.4 层次遍历

除了前面介绍的 3 种遍历方法外，二叉树还可以层次遍历，其过程是：

若二叉树非空，则

step1：访问根结点(第一层)。

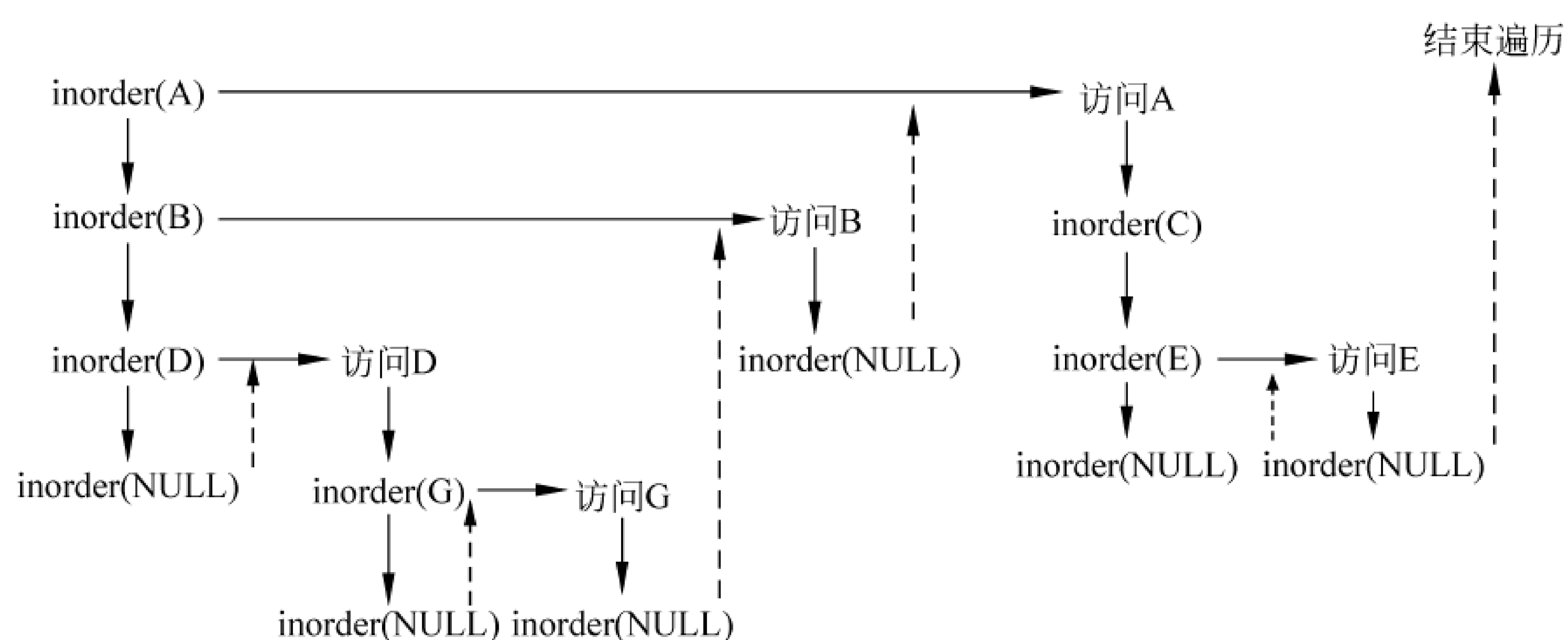
step2：逐层向下访问结点。

step3：同一层上的结点从左向右访问。

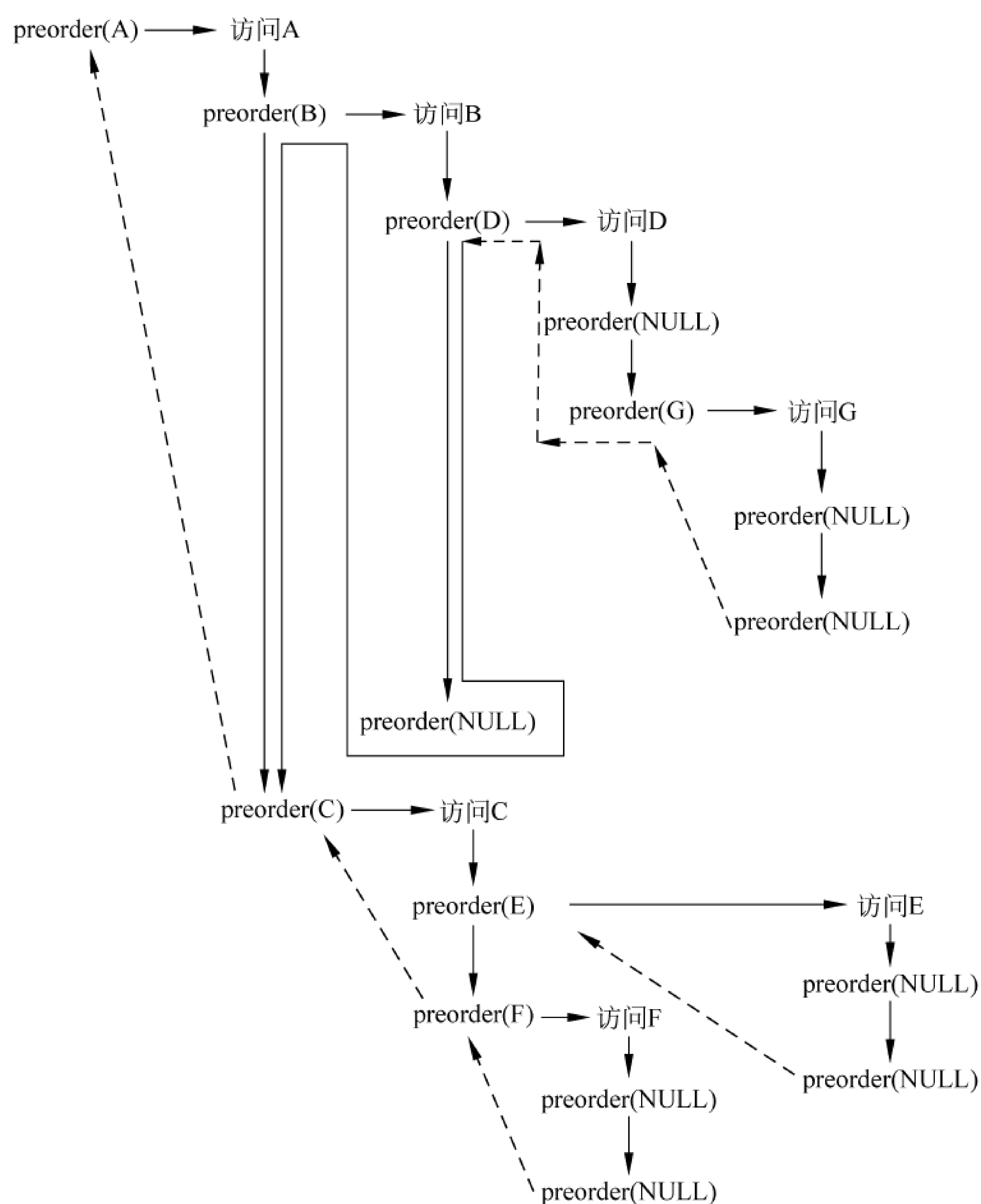
例如，图 6.9(a)的二叉树的层次序列为：ABCDEFG。

二叉树的中序、先序、后序 3 种遍历递归过程如图 6.11 所示。具体算法如下。

```
void  inorder(BiTree * root)              //已知二叉树 root
{  if(root!=NULL)
   {
     inorder(root->lchild);                //中序遍历左子树
     printf(root->data);                   //访问根结点
     inorder(root->rchild);                //中序遍历右子树
   }
}
void  preorder(BiTree * root)             //已知二叉树 root
{  if(root!=NULL)
  {
    printf(root->data);                    //访问根结点
    preorder(root->lchild);                //先序遍历左子树
    preorder(root->rchild);                //先序遍历右子树
  }
}
void  postorder(BiTree * root)            //已知二叉树 root
{  if(root!=NULL)
  {
    postorder(root->lchild);               //后序遍历左子树
    postorder(root->rchild);               //后序遍历右子树
    printf(root->data);                    //访问根结点
  }
}
```

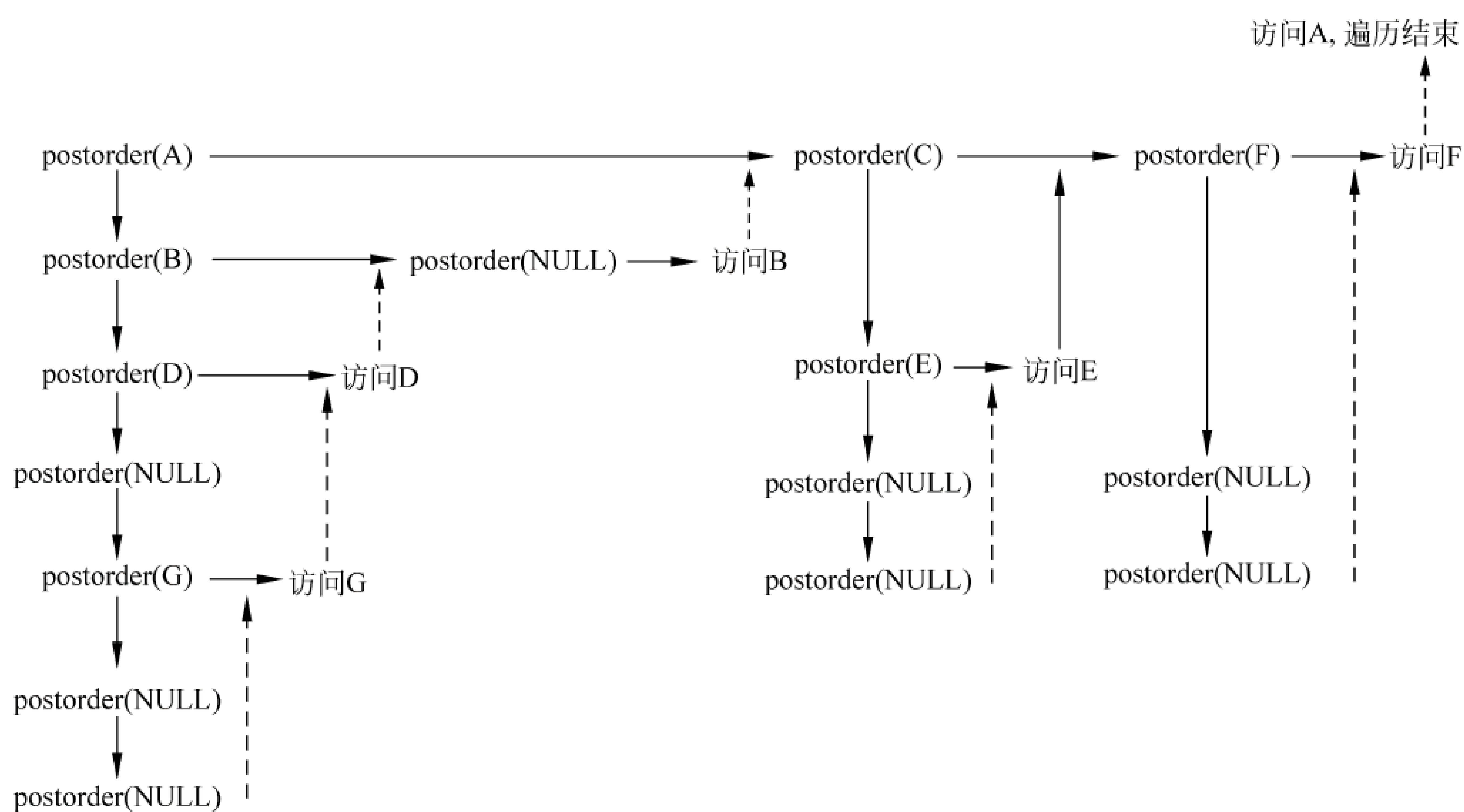


(a) 中序遍历递归过程



(b) 先序遍历递归过程

图 6.11　3 种遍历递归过程



(c) 后序遍历递归过程

图 6.11 (续)

层次遍历过程无法递归实现,但可以通过队列实现层次遍历。先将根结点入队列,在队列不为空时循环:从队列中出队列一个结点 p,访问它;若它有左孩子结点,则将其左孩子结点入队列;若它有右孩子结点,则将其右孩子结点入队列。如此重复操作,直到队空为止。对应的算法如下。

```
void  Levelorder(BiTree  * root)
{ BTNode  * p, * Q[MaxSize];                    //定义结点指针 p、环形队列 Q
  int  front=-1, rear=-1;
  rear=rear+1;
  Q[rear]=root;                                 //根结点入队列
  while(front!=rear)                            //队列不为空
  { front=(front+1)%MaxSize;
   p=Q[front];                                  //结点出队列
   printf(p->data);                             //访问结点
   if(p->lchild!=NULL)                          //访问的当前结点有左孩子
   {
     rear=(rear+1)%MaxSize;
     Q[rear]=p->lchild;}
   }
   if(p->rchild!=NULL)                          //访问的当前结点有右孩子
   {
     rear=(rear+1)%MaxSize;
     Q[rear]=p->rchild;
   }
  }
}
```

step1：根结点 A 入队列。

step2：判断队列是否为空，若不空，则队首元素出队列。

step3：指针 p 指向出队列的结点，输出该结点，并将其左孩子结点、右孩子结点入队列。

step4：重复步骤 **step2** 和 **step3**，直至队列为空。

```
根结点 A 入队列；
while(判断队列是否为空)
{队首元素出队列；
指针 p 指向出队列的结点；
输出该结点；
并将其左孩子结点、右孩子结点依次入队列；
  }
```

- A 入队列，A 出队列，输出 A，A 的左、右孩子 B、C 入队列。
- B 出队列，输出 B，B 的左孩子 D 入队列。
- C 出队列，输出 C，C 的左、右孩子 E、F 入队列。
- D 出队列，输出 D，D 的右孩子 G 入队列。
- E 出队列，输出 E，E 无左、右孩子入队列。
- F 出队列，输出 F，F 无左、右孩子入队列。
- G 出队列，输出 G，G 无左、右孩子入队列。

队列为空，算法结束。

【例 6.8】 已知一棵二叉树的前序遍历结果是 ABECDFGHIJ，中序遍历结果是 EBCDAFHIGJ，试画出这棵二叉树。

分析：根据二叉树前序遍历规则“根左右”可以找到根结点 A；根据中序遍历规则“左根右”可以由根结点 A 分出哪些结点构成左子树(EBCD)哪些结点构成右子树(FHIGJ)。同样，左、右子树对应的二叉树先根据先序遍历法则找出根结点，再根据中序遍历法则分出其左、右子树，以此类推，直至画出该二叉树。二叉树的构造过程如图 6.12 所示。

注意：由先序序列＋中序序列或者中序序列＋后序序列可唯一确定一棵二叉树。

【例 6.9】 表达式 23＋(12＊3－2)/4＋34/7 的后缀表达式是什么？

分析：后缀表达式指的是不包含括号，运算符放在两个运算对象的后面，所有的计算按运算符出现的顺序严格从左向右进行(不再考虑运算符的优先规则)。由表达式计算后缀表达式，首先构建表达式的语法树，对语法树进行后序遍历即可得到后缀表达式。表达式 23＋(12＊3－2)/4＋34/7 对应的语法树如图 6.13 所示。

对该表达式对应的语法树进行后序遍历，得到后缀表达式 23 12 3 ＊ 2 － 4 / ＋ 34 7 / ＋。

6.3.5 二叉树遍历的应用


视频讲解

二叉树的递归遍历不仅可以得到关于结点的线性序列，而且可以利用这种递归遍历的思想实现二叉树的其他基本操作。

1. 创建二叉树

根据先序遍历序列对应的字符串建立二叉树的递归算法。注意，此先序序列是“原生态”的。例如，创建如图 6.10(a)所示的二叉树，其先序序列为“ABD□G□□□CE□□F□□”，其

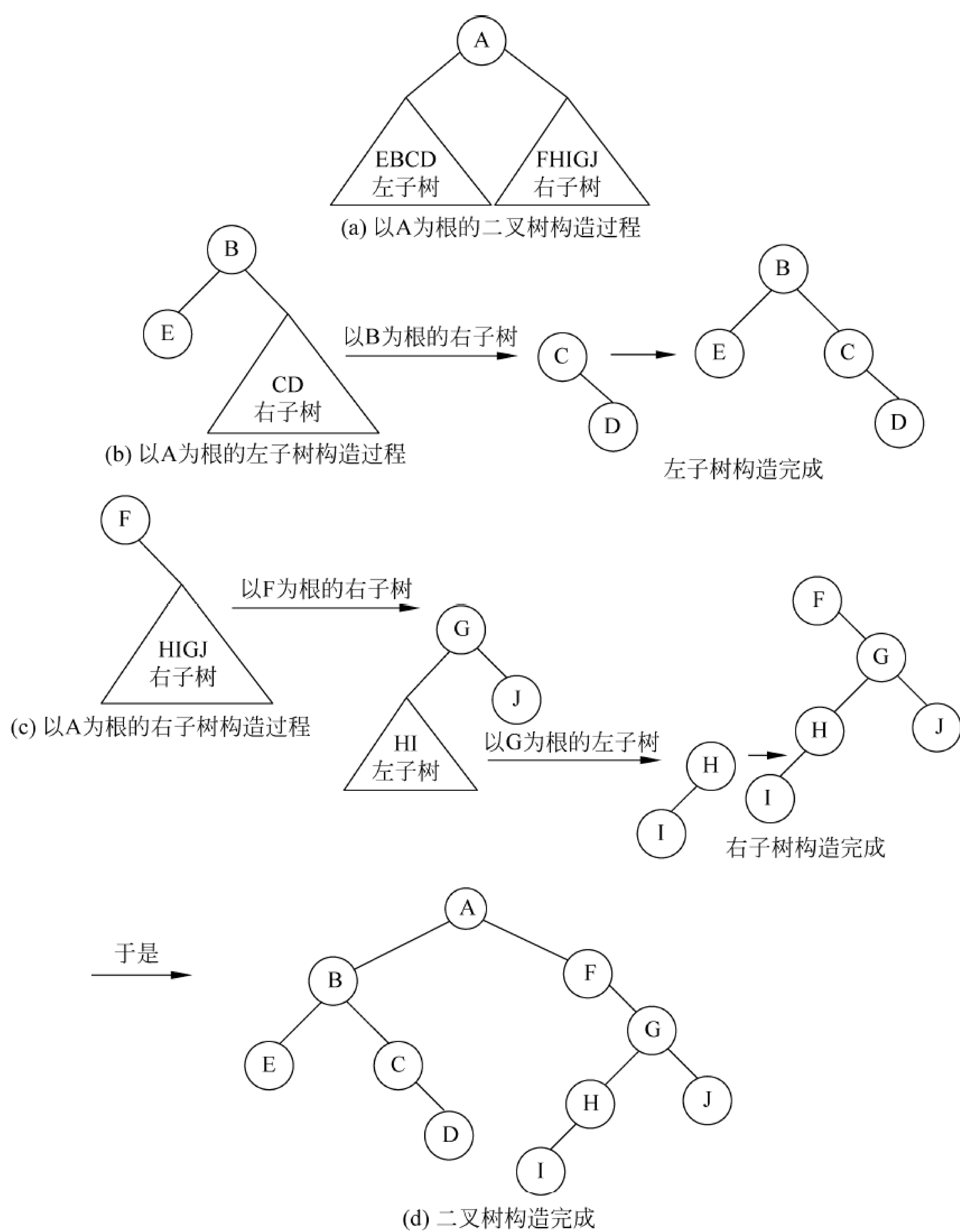


图 6.12　二叉树的构造过程

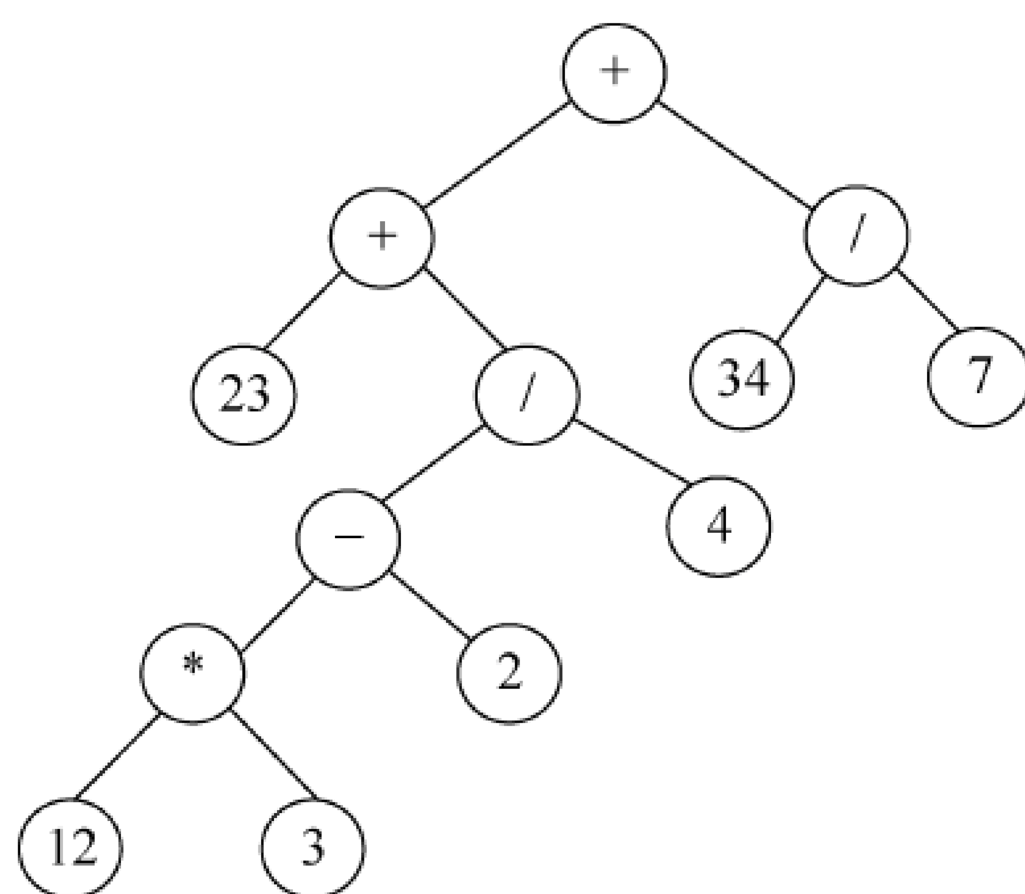


图 6.13　表达式 23＋(12＊3－2)/4＋34/7 对应的语法树

中“□”为空格,表示空结点。模拟先序遍历方式,先创建根结点,然后先序创建左子树,最后先序创建右子树。算法实现如下。

```
void  CreatBitree(BiTree * &root, char * str)                    //先序序列对应的字符串 str
{  char  ch= * str;
   if(ch='□')
   root=NULL;                                                   //root 为空树
   else
   {
   root=(BiTree * )malloc(sizeof(BTNode));
   root->data=ch;
   CreatBitree(root->lchild,str++);
   CreatBitree(root->rchild,str++);
}
}
```

2. 计算二叉树的高度

根据二叉树的先序、中序、后序遍历可知,算法实现时将二叉树分成了 3 部分:根结点、左子树、右子树;因此,模拟这种方式计算二叉树高度时,也可以将其分成 3 部分,这样高度的计算模型为

$$h = \begin{cases} 0 & \text{空树} \\ \text{左、右子树较高者} + 1 & \text{非空树} \end{cases}$$

算法实现如下。

```
int  Height(BiTree  * root)
{
int  hL=0, hR=0, h=0;
if(root==NULL)
   return  h;
else
{
   hL=Height(root->lchild);
   hR=Height(root->rchild);
   return  (hL>=hR? hL+1:hR+1);
  }
}
```

3. 计算二叉树的结点个数

从二叉树高度的计算可知,二叉树的结点个数等于左子树结点个数+右子树结点个数+根结点个数。

```
int  CountNode(BiTree * root)
{ int  nL=0,nR=0, n=0;
if(root==NULL)
   return  n;
```

```
  else
  {
     nL=CountNode(root->lchild);
     nR=CountNode(root->rchild);
     return  (nL+nR+1);
   }
  }
```

同时,根据二叉树遍历的递归算法还可以实现“输出二叉树的叶子结点,计算叶子结点个数,二叉树的左、右子树互换,复制二叉树等”。


视频讲解

6.3.6 二叉树遍历的非递归实现

1. 中序遍历的非递归算法

递归算法实现二叉树的遍历,程序编辑虽然简单,但是程序的运行过程(递归调用的逐层调用和逐层返回过程)相当麻烦,尤其是当树的规模较大时,算法的效率较低,因此将递归算法转化为非递归算法是提高算法效率的一种必要手段。递归算法转化为非递归算法优先考虑到栈。

由中序遍历的过程可知,中序序列的开始结点是一棵二叉树的最左下结点,其基本思路是:先找到二叉树的开始结点,访问它,再处理其右子树。由于二叉链表中指针的方向是单向的,因此采用一个栈保存需要返回的结点的指针。

算法过程:用指针指向当前要处理的结点,先扫描(并非访问)根结点所有左结点,并将它们一一进栈,当无左结点时,表示栈顶结点无左子树,然后出栈这个结点,并访问它,将 p 指向刚出栈的结点的右孩子,对右子树进行同样的处理。需要注意的是,当结点 * p 的所有左下结点进栈后,这时的栈顶结点要么没有左子树,要么其左子树已经访问过,就可以访问这个栈顶结点。如此重复操作,直到栈空为止。对应的算法如下。

```
  void InOrder1(BiTree * root)          //中序非递归遍历算法
  {
      BTNode * St[MaxSize], * p;
      int top=-1;
      if (root!=NULL)
      {p=root;
      while (top>-1 || p!=NULL)  //处理 * b 结点的左子树
      {
       while (p!=NULL)               //扫描 * p 的所有左结点并进栈
       {
        top++;
        St[top]=p;
        p=p->lchild;
       }
                                       //执行到此处时,栈顶元素没有左孩子或左子树均已访问过
      if (top>-1)
      {
```

```
            p=St[top];                    //出栈 * p 结点
            top--;
            printf("%c ",p->data);        //访问之
            p=p->rchild;                  //扫描 * p 的右孩子结点
        }
    }
    printf("\n");
    }
}
```

对于图 6.10(b)的二叉树 root,执行上述算法时,栈的操作过程及其说明见表 6.1,最后的输出序列为 DGBAECF。

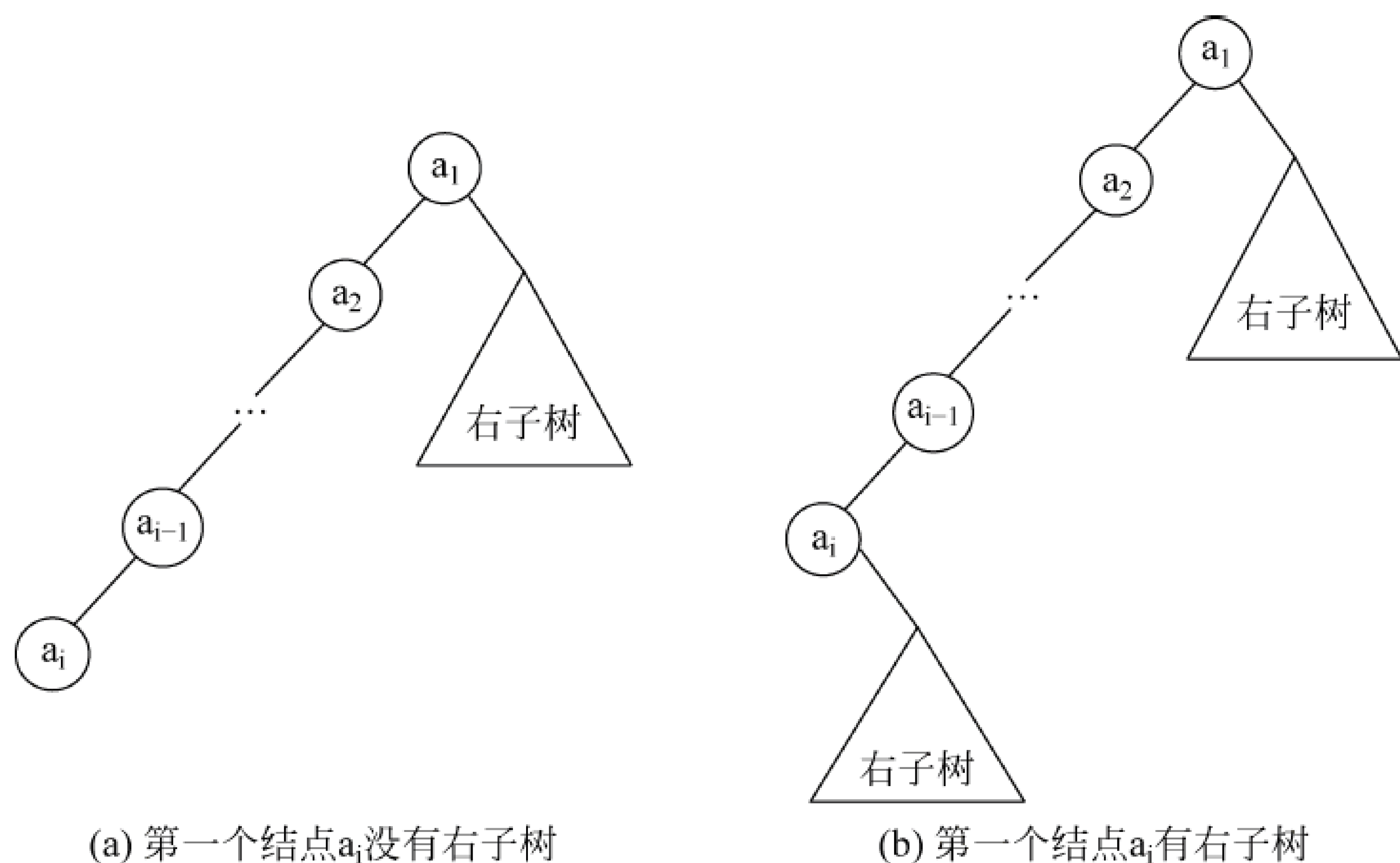
表 6.1　Inorder1(root)执行时栈的操作过程及其说明

操　作	结　点	top	栈中结点	说　明
进栈	A	0	A	根结点 A 进栈
进栈	B	1	AB	根结点 A 的左孩子 B 进栈
进栈	D	2	ABD	结点 B 的左孩子 D 进栈
出栈	D	1	AB	栈顶结点 D 没有左孩子,D 出栈,访问 D,指针 p 移到 D 的右孩子 G
进栈	G	2	ABG	结点 G 进栈
出栈	G	1	AB	结点 G 没有左孩子,G 出栈,访问 G,p 指向 G 的左、右孩子⇒p=NULL
出栈	B	0	A	栈顶结点 B 的左子树已访问,B 出栈,访问 B,p 指向 B 的右孩子⇒p=NULL
出栈	A	−1	空	栈顶结点 A 的左子树已经访问,A 出栈,访问 A,p 指向 A 的右孩子 C
进栈	C	0	C	结点 C 进栈
进栈	E	1	CE	结点 C 的左孩子 E 进栈
出栈	E	1	C	栈顶结点 E 没有左孩子,E 出栈,访问 E,p 指向 E 的右孩子⇒p=NULL
出栈	C	−1	空	结点 C 的左子树已访问,C 出栈,访问 E,p 指向 G 的右孩子 F
进栈	F	0	F	结点 F 没有左孩子,F 出栈,访问 F,p 指向 F 的右孩子⇒p=NULL
出栈	F	−1	空	栈空且 p=NULL,退出循环,算法结束

除了使用栈外,中序遍历的非递归算法还可以依靠算法本身实现,按照一种“纯手工”的方式,将结点一个一个遍历输出。首先分析中序遍历的第一个结点,情况如图 6.14 所示。

从根结点出发,沿着左分支一直走到头(找到第一个没有左孩子的结点),最左下方的结点 a_i 为中序遍历的第一个点;随后在访问第二个结点时,就要分两种情况讨论:如图 6.14(a)所示,若 a_i 结点没有右子树时,中序遍历的第二个结点应向上找,即 a_{i-1} 结点,而二叉链表中并没有向上的指针域,所以不借助任何方式遍历 a_{i-1} 结点非常困难,这就引入了后面的知识

点——线索；另一种情况如图 6.14(b)所示，若 a_i 结点存在右子树时，中序遍历的第二个结点应该在其右子树上，且是其右子树中序遍历的第一个结点，而关于第一个结点的查找过程前面已经说明过，在此不再赘述；以此类推，直到遍历到最后一个结点为止。



(a) 第一个结点a_i没有右子树　　(b) 第一个结点a_i有右子树

图 6.14　中序遍历某二叉树

2. 先序遍历非递归算法

先序遍历过程：先访问根结点，再访问左子树，最后访问右子树。因此，先将根结点进栈，在栈不为空时循环出栈 p，再访问 p 结点，若其右孩子结点不为空，则将右孩子结点进栈，若其左孩子不为空，再将其左孩子结点进栈。对应的算法如下。

```
void  PreOrder1(BiTree  * root)                //先序非递归遍历算法
{
    BTNode * St[MaxSize], * p;
    int top=-1;
    if (root!=NULL)
    {
        top++;                                 //根结点进栈
        St[top]=root;
        while (top>-1)                         //栈不为空时循环
        {
            p=St[top];                         //退栈并访问该结点
            top--;
            printf("%c ",p->data);
            if (p->rchild!=NULL)               //右孩子结点进栈
            {
                top++;
                St[top]=p->rchild;
            }
            if (p->lchild!=NULL)               //左孩子结点进栈
            {
                top++;
```

```
                    St[top]=p->lchild;
                }
            }
            printf("\n");
        }
    }
```

对于图6.10(b)的二叉树root,执行上述算法时,栈的操作过程及其说明见表6.2,最后的输出序列为ABDGCEF。

表6.2 PreOrder1(root)执行时栈的操作过程及其说明

操作	结点	top	栈中结点	说明
进栈	A	0	A	根结点A进栈
出栈	A	−1	空	根结点A出栈,访问A
进栈	C	0	C	将根结点的右孩子C进栈
进栈	B	1	CB	将根结点的左孩子B进栈
出栈	B	0	C	结点B出栈访问B
进栈	D	1	CD	结点B的左孩子进栈
出栈	D	0	C	结点D出栈,访问D
进栈	G	1	CG	结点D的右孩子进栈
出栈	G	0	C	结点G出栈,访问C
出栈	C	−1	空	结点C出栈,访问C
进栈	F	0	F	结点C的右孩子F进栈
进栈	E	1	FE	结点C的左孩子E进栈
出栈	E	0	F	结点E出栈,访问E
出栈	F	−1	空	结点F出栈,访问F,栈空算法结束

3. 后序遍历非递归算法

在后序遍历中,对于每个结点,先访问其左子树,然后访问其右子树,最后才访问该结点本身。与中序遍历情况类似,后序遍历中第一个访问的结点是二叉树的最左下结点。由于首先访问结点的左、右子树,之后才访问结点本身,所以对于任一结点,必须知道其左、右子树是否被访问过。

采用一个栈保存返回的结点指针,先扫描根结点的所有左孩子结点并一一进栈,出栈一个结点b为当前结点,然后扫描该结点的右子树。当一个结点的左、右孩子结点均被访问后再访问该结点,如此重复操作,直到栈空为止。

其中的难点是如何判断一个结点b的右子树已经被访问过(实际上,若右孩子结点已被访问过,其右子树就已经被访问过),为此用p指针保存刚刚被访问过的结点(初值为NULL),若b->rchild==p成立(在后序遍历中,b的右孩子结点一定刚好在访问b之前被访问),说明b的左、右孩子均已被访问,现在应访问b。

从上述过程可知,栈中保存的是当前结点b的所有祖先结点(均未被访问过)。对应的算法如下。

```
void  PostOrder1(BiTree  * root)                    //后序非递归遍历算法
{
    BTNode * St[MaxSize], * b=root;
    BTNode * p;
    int flag, top=-1;                               //栈指针置初值
    if (b!=NULL)
    {
        do
        {
            while (b!=NULL)                         //将 * b 的所有左结点进栈
            {
                top++;
                St[top]=b;
                b=b->lchild;
            }
            //执行到此处时,栈顶元素没有左孩子或左子树均已被访问过
            p=NULL;                                 //p 指向栈顶结点的前一个已被访问的结点
            flag=1;                                 //设置 b 的访问标记为已访问过
            while (top!=-1 && flag)
            {    b=St[top];                         //取出当前的栈顶元素
                 if (b->rchild==p)
                 {
                     printf("%c ",b->data);         //访问 b 结点
                     top--;
                     p=b;                           //p 指向刚访问过的结点
                 }
                 else
                 {
                     b=b->rchild;                   //b 指向右孩子结点
                     flag=0;                        //设置未被访问的标记
                 }
            }
        } while (top!=-1);
        printf("\n");
    }
}
```

对于图 6.10(b)的二叉树 root,执行上述算法时的操作过程及其说明见表 6.3,最后的输出序列为 GDBEFCA。

表 6.3　PostOrder1(root)执行时栈的操作过程及其说明

操作	结点	top	栈中结点	说　明
进栈	A	0	A	根结点 A 进栈
进栈	B	1	AB	根结点 A 的左孩子 B 进栈
进栈	D	2	ABD	结点 B 的左孩子 D 进栈
进栈	G	3	ABDG	结点 D 的右孩子 G 进栈
出栈	G	2	ABD	p=NULL,flag=1,b 指向栈顶结点 G,G 的右孩子为空(b->rchild==p 成立),G 出栈,访问 G,p=b 指向 G

续表

操作	结点	top	栈中结点	说　明
出栈	D	1	AB	p 指向 G，flag＝1，b 指向栈顶结点 D，D 的右孩子为 G(b→rchild＝＝p 成立)，D 出栈，访问 D，p＝b 指向 D
出栈	B	0	A	p 指向 D，flag＝1，b 指向栈顶结点 B，B 的右孩子为空(b→rchild＝＝p 不成立)，b 指向 B 的右孩子(b＝NULL，flag＝0，第一个 while 条件不成立，p＝NULL，flag＝1，b 指向栈顶结点 B，B 的右孩子为空(b→rchild＝＝p 成立)，B 出栈，访问 B，p＝b 指向 B
进栈	C	1	AC	b 指向结点 C，C 进栈
进栈	E	2	ACE	结点 C 的左孩子 E 进栈
出栈	E	1	AC	p＝NULL，flag＝1，b 指向结点 E，E 的右孩子为空(b→rchild＝＝p 成立)，E 出栈，访问 E，p＝b 指向 E
进栈	F	2	ACF	b 指向结点 F，F 进栈
出栈	F	1	AC	p＝NULL，flag＝1，b 指向栈顶结点 E，E 的右孩子为空(b→rchild＝＝p 成立)，F 出栈，访问 F，p＝b 指向 F
出栈	C	0	A	p 指向 F，b 指向栈顶结点 C，C 的右孩子为 F(b→rchild＝＝p 成立)，C 出栈，访问 C，p＝b 指向 C
出栈	A	－1	空	p 指向 C，b 指向栈顶结点 A，A 的右孩子为 C(b→rchild＝＝p 成立)，A 出栈，访问 A，栈空，算法结束

以上后序非递归遍历算法有这样的特点：访问某个结点时，栈中保存的正好是该结点的所有祖先结点，从栈顶到栈底正好是该结点的双亲结点到根结点路径上的结点序列。有些复杂的算法就是利用这个特点设计的。

【例 6.10】 假设二叉树采用二叉链存储结构，请设计一个算法，输出从根结点到每个叶子结点的路径的逆(树中路径是从根结点到其他结点的结点序列，而这里是求从叶子结点到根结点的序列，或者说是求叶子结点及其所有祖先结点的序列)。本例要求采用后序遍历非递归算法。

解：利用后序遍历非递归算法的特点将算法中访问结点的操作改为判断该结点是否为叶子结点，若是，则输出栈中的所有结点值。对应的算法如下。

```
void  AllPath1(BTNode *b)
{
    BTNode *St[MaxSize];
    BTNode *p;
    int flag,i,top=-1;                          //栈指针置初值
    if (b!=NULL)
    {
    do
    {   while (b!=NULL)                         //将 b 的所有左结点进栈
      { top++;
        St[top]=b;
        b=b->lchild;
        }
```

```
            p=NULL;                          //p指向栈顶结点的前一个已访问的结点
            flag=1;                          //设置b的访问标记为已访问过
            while (top!=-1 && flag)
            {   b=St[top];                   //取出当前的栈顶元素
                if (b->rchild==p)
                  { if (b->lchild==NULL && b->rchild==NULL)
                                             //若为叶子结点
                      {  //输出栈中的所有结点值
                         for (i=top;i>0;i--)
                            printf("%c->",St[i]->data);
                         printf("%c\n",St[0]->data);
                      }
                    top--;
                    p=b;                     //p指向刚访问过的结点
                  }
               else
                  { b=b->rchild;             //b指向右孩子结点
                    flag=0;                  //设置未被访问的标记
                  }
            }
        } while (top!=-1);
    printf("\n");
        }
    }
```

对于图6.10(a)所示的二叉树,其存储结构为G→D→B→A、E→C→A和F→C→A这3条路径序列。

【例6.11】 采用层次遍历方法设计算法。

解:这里设计的队列为非环形顺序队列qu,将所有已访问过的结点指针进队,并在队列中保存双亲结点的位置。当找到一个叶子结点时,在队列中通过双亲结点的位置输出根结点到该叶子结点的路径的逆。

```
    void AllPath2(BTNode *b)
    {
        struct snode
        {
            BTNode *node;                        //存放当前结点指针
            int parent;                          //存放双亲结点在队列中的位置
        } qu[MaxSize];                           //定义非环形队列
        BTNode *q;
        int front,rear,p;                        //定义队头和队尾指针
        front=rear=-1;                           //置队列为空队列
        rear++;
        qu[rear].node=b;                         //根结点指针进入队列
        qu[rear].parent=-1;                      //根结点没有双亲结点
        while (front!=rear)                      //队列不为空
```

```
    {
        front++;
        q=qu[front].node;                          //队头出队列,该结点指针仍在 qu 中
        if (q->lchild==NULL && q->rchild==NULL)    //q 为叶子结点
        {
            p=front;
            while (qu[p].parent!=-1)
            {       printf("%c->",qu[p].node->data);
                    p=qu[p].parent;
            }
            printf("%c\n",qu[p].node->data);
        }
        if (q->lchild!=NULL)                       //q 结点有左孩子时将其进列
        {   rear++;
            qu[rear].node=q->lchild;
            qu[rear].parent=front;
        }
        if (q->rchild!=NULL)                       //q 结点有右孩子时将其进列
        {   rear++;
            qu[rear].node=q->rchild;
            qu[rear].parent=front;
        }
    }
}
```

对于图 6.10(a)所示的二叉树 b,执行 AllPath2(b)后,队列 qu 中的元素情况见表 6.4,当访问到值为 G 的结点时(其 foot=6),判定它是叶子结点,输出路径的过程是:输出 G,通过 parent=3 找到双亲结点值为 D(其 foot=3),输出 D;通过 parent=1 找到双亲结点值为 B(其 foot=1),输出 B;通过 parent=0 找到双亲结点的值为 A(其 foot=0),输出 A;由于结点 A 的 parent=-1,因此本次输出路径终止。

表 6.4　指向 AllPath2(b)后队列 qu 中的元素情况

front	qu[front]所指结点值	qu[front].parent(当前双亲结点的位置)
0	A	-1
1	B	0
2	C	0
3	D	1
4	E	2
5	F	2
6	G	3

6.4 线索二叉树



视频讲解

6.4.1 线索二叉树的概念

二叉树遍历是其最重要的运算之一，常常采用递归算法实现。将递归算法转化为非递归算法常常借助栈实现，6.3 节中已经做了详细描述。除了使用栈实现非递归的遍历，二叉树也可以从自身着手，为遍历的实现做出准备。

对于具有 n 个结点的二叉树，采用二叉链表存储结构时，每个结点都有两个指针域，总共有 2n 个指针域，又由于只有 n－1 个结点被有效指针指向(n 个结点中只有树根结点没有被有效指针指向)，则共有 2n－(n－1)＝n＋1 个空链域。显然，这些空指针域就被浪费掉了。

遍历二叉树的结果是一个结点的线性序列。可以利用这些空链域存放指向结点前驱结点和后继结点的指针。这样的指向某线性序列中的“前驱结点”和“后继结点”的指针被称为**线索**。借助某种遍历结果为二叉树加上线索的过程称为**线索化**。二叉树常见的 3 种遍历为先序遍历、中序遍历、后序遍历，于是就相应地产生**先序线索二叉树**、**中序线索二叉树**、**后序线索二叉树**。线索化的本质既将空指针域由空到非空利用起来，同时又为二叉树的遍历提供了方便。

由于遍历方式不同，因此产生遍历线性序列也不同，可做如下规定：当某结点的左指针 lchild 为空时，令该指针指向按某种方式遍历二叉树时得到的该结点的前驱结点；当某结点的右指针 rchild 为空时，令该指针指向按某种方式遍历二叉树得到的该结点的后继结点。但如何区分左指针指向的结点到底是左孩子结点，还是前驱结点；右指针指向的结点到底是右孩子结点，还是后继结点呢？在结点的存储结构上增加两个标志位可区分这两种情况。

$$\text{左标志 ltag}=\begin{cases}0 & \text{表示 lchild 指向左孩子结点}\\1 & \text{表示 lchild 指向前驱结点}\end{cases}$$

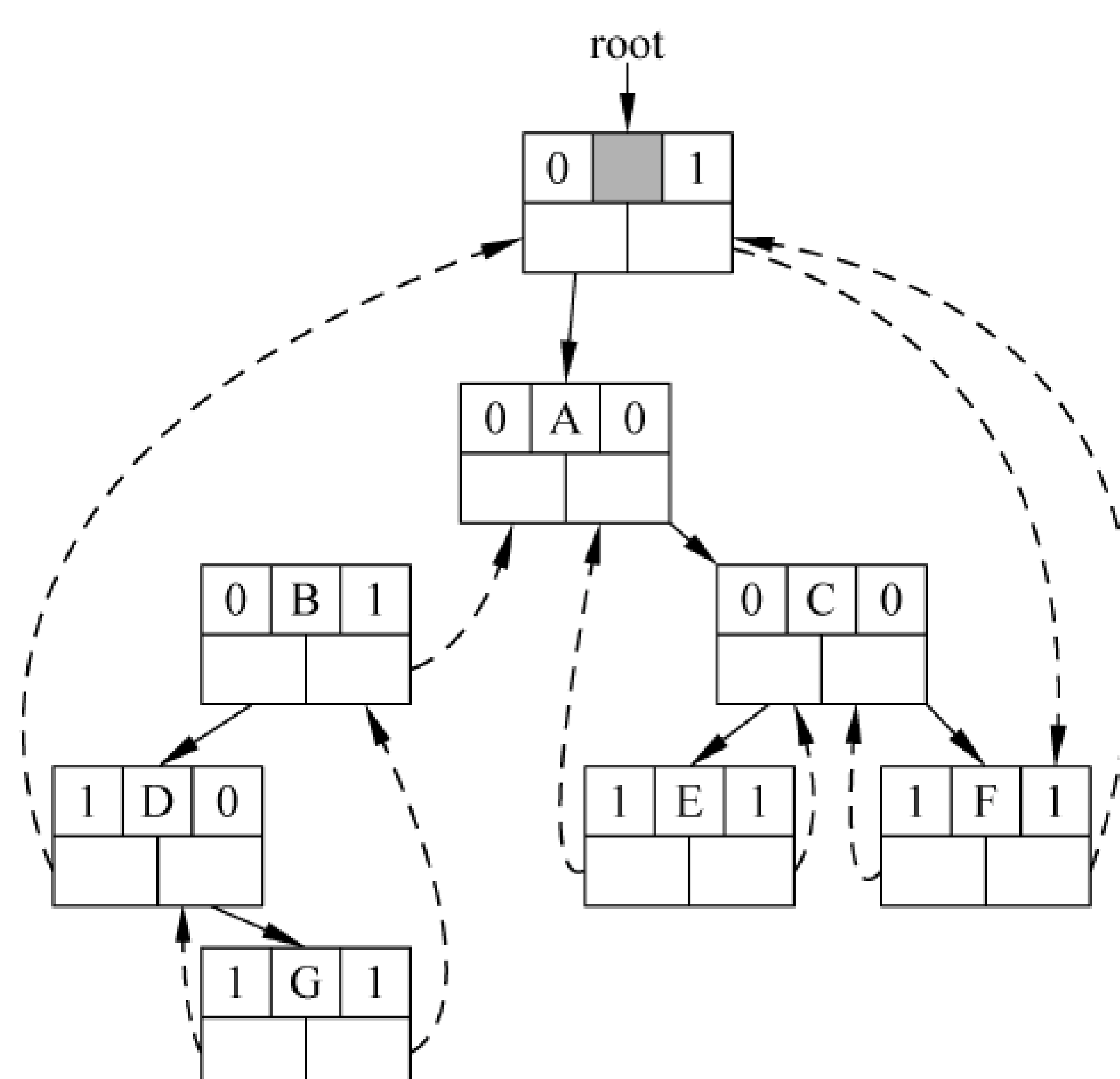
$$\text{右标志 rtag}=\begin{cases}0 & \text{表示 rchild 指向右孩子结点}\\1 & \text{表示 rchild 指向后继结点}\end{cases}$$

这样，每个结点的存储结构如下。

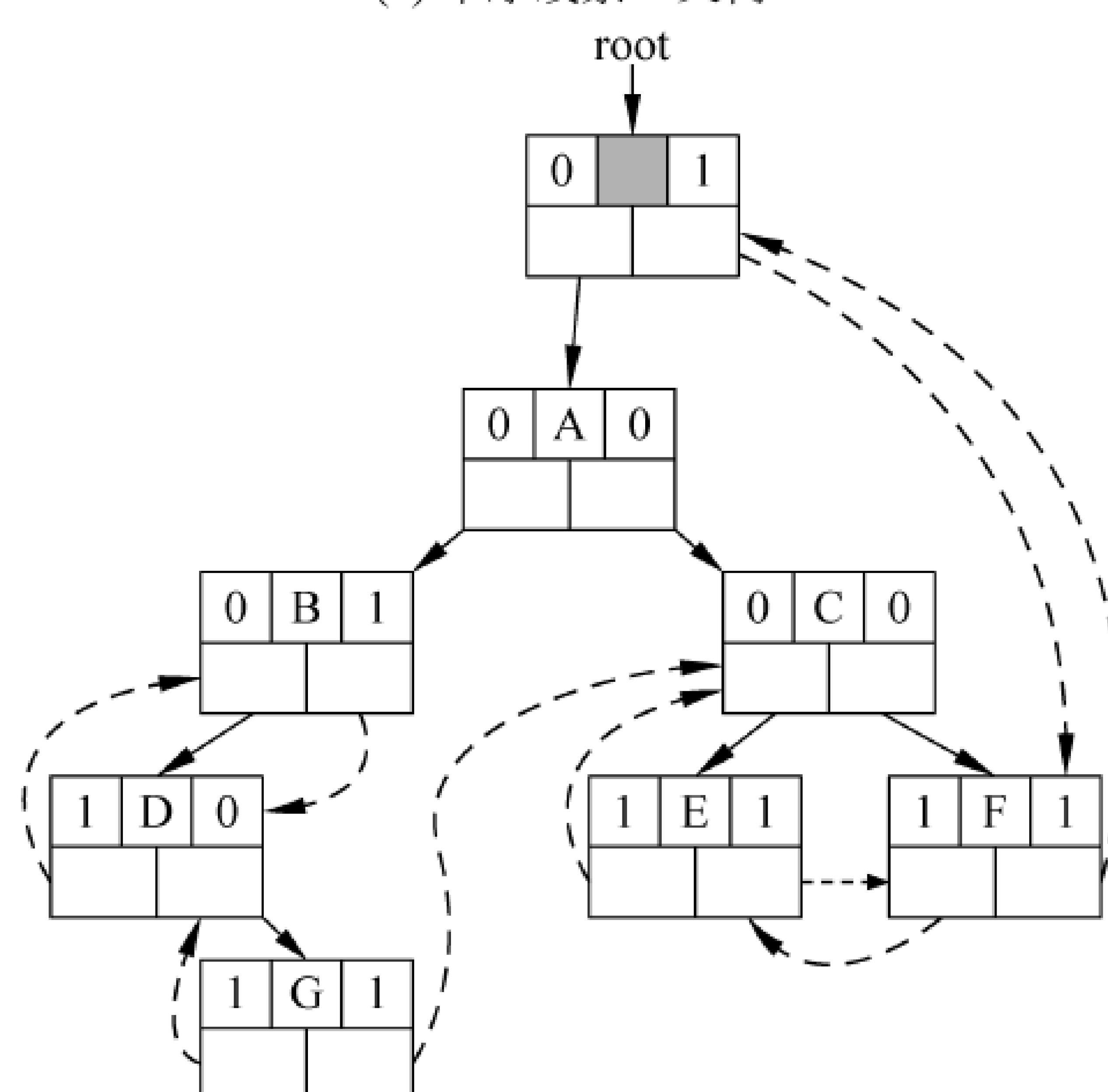
lchild	ltag	data	rtag	rchild

按上述原则在二叉树的每个结点上加上线索的二叉树称作**线索二叉树**。对二叉树以某种方式进行遍历，使其变为线索二叉树的过程称为对二叉树进行**线索化**。

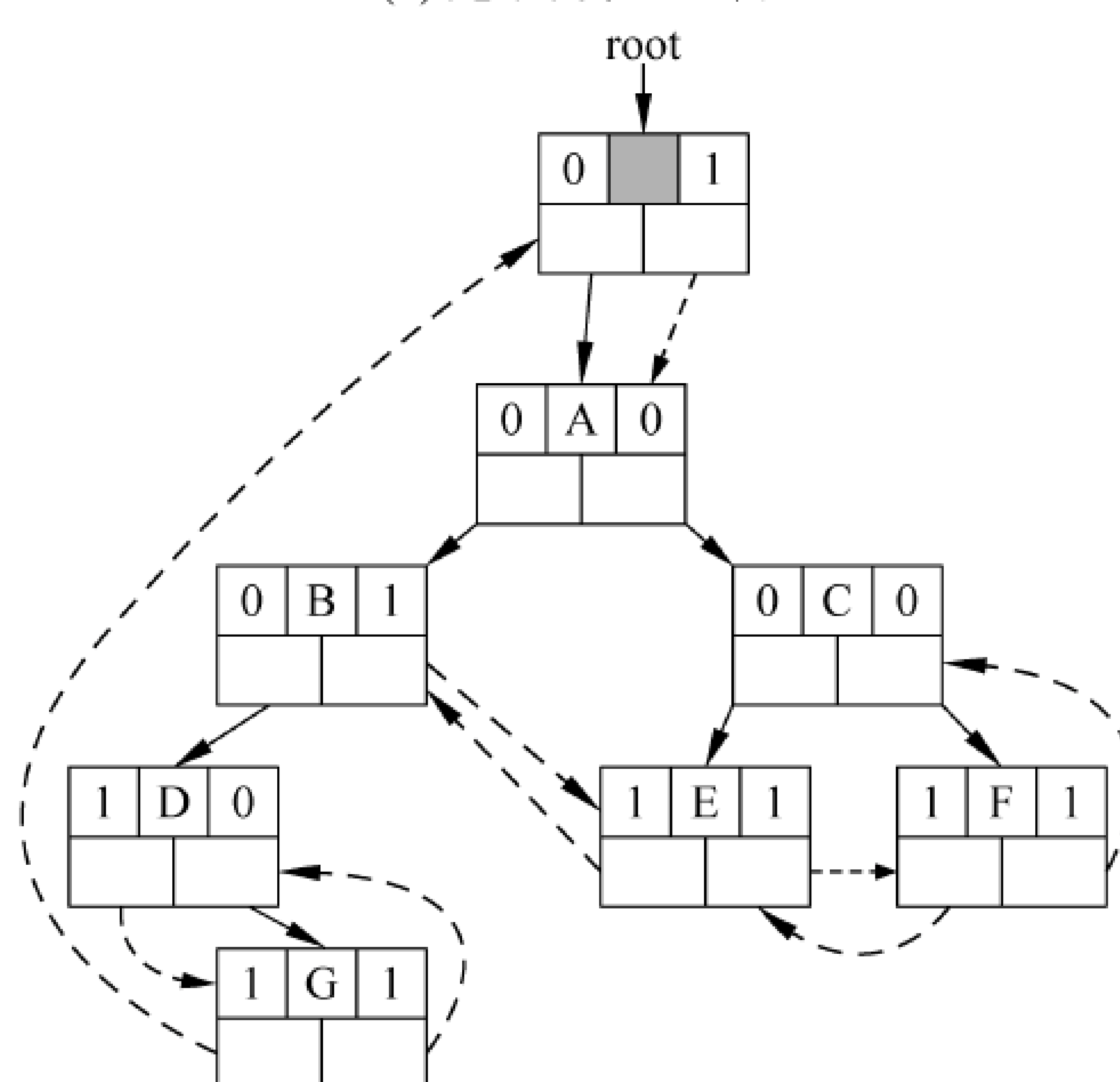
为使算法设计方便，可在线索二叉树中再增加一个头结点。头结点的 data 域为空；lchild 指向二叉树的根结点，ltag 为 0；rchild 指向按某种方式遍历二叉树时的最后一个结点，rtag 为 1。图 6.15 为图 6.10 所示二叉树的线索二叉树。其中，图 6.15(a)是中序线索二叉树(中序序列为 DGBAECF)，图 6.15(b)是先序线索二叉树(先序序列为 ABDGCEF)，图 6.15(c)是后序线索二叉树(后序序列为 GDBEFCA)。图 6.15 中，实线表示二叉树原来指针指向的结点，虚线表示线索二叉树所添加的线索。



(a) 中序线索二叉树



(b) 先序线索二叉树



(c) 后序线索二叉树

图 6.15　线索二叉树

注意：

- 在中序、先序和后序线索二叉树中，所有实线均相同，即线索化之前的二叉树相同，所有结点的标志位取值也完全相同，只是当标志位取 1 时，不同的线索二叉树将用不同的虚线，即不同的线索树中线索指向的前驱结点和后继结点不同。
- n 个结点的线索二叉树共有 n+1 个线索。
- 二叉树线索化的目的是方便实现二叉树非递归遍历。

6.4.2 线索化二叉树

从 6.4.1 节的讨论得知：遍历同一棵二叉树的方式不同，所得到的线索二叉树也不同。二叉树有先序、中序和后序 3 种遍历方式，所以线索二叉树也有先序线索二叉树、中序线索二叉树和后序线索二叉树 3 种。下面以中序线索二叉树为例，讨论建立线索二叉树的算法。

建立线索二叉树，或者说，对二叉树线索化，实质上就是遍历一棵二叉树，在遍历的过程中，检查当前结点的左、右指针域是否为空。如果为空，则将它们改为指向前驱结点或后继结点的线索。为了记录遍历过程中访问结点的先后次序，可附加一个指针 pre 始终指向刚刚访问过的结点，若指针 p 指向当前访问的结点，则 pre 指向它的前驱结点。于是可得到中序遍历，建立中序线索化链表。

为了实现二叉树线索化，将前面二叉树结点的类型定义修改如下。

```
    typedef struct TBTNode
{   elemtype data;                          //结点数据域
    int ltag,rtag;                          //增加的线索标记
    struct TBTNode *lchild;                 //左孩子或前驱线索的指针
    struct TBTNode *rchild;                 //右孩子或后继线索的指针
}TBiTree;                                   //线索二叉树类型 TBiTree 的定义
```

下面是建立中序线索二叉树的算法。CreaThread(b)算法是将以二叉链表存储的二叉树 b 进行中序线索化，并返回线索化后头结点的指针 root。Thread(p)算法用于对以 *p 为根结点的二叉树中序线索化。在整个算法中，指针 p 总是指向当前被线索化的结点，而 pre 指向刚刚访问过的结点，*pre 是 *p 的前驱结点，*p 是 *pre 的后继结点。

CreaThread(b)算法的思路是：先创建头结点 *root，其 lchild 域为链指针，rchild 域为线索；将 lchild 指针指向 *b，如果二叉树 b 为空，则将 lchild 指向自身，否则将 *root 的 lchild 指向 *b 结点，p 指向结点 *b，pre 指向结点 *root；再调用 Thread(b)对整个二叉树线索化；最后加入指向头结点的线索，并将头结点的 rchild 指针域线索化为指向最后一个结点(由于线索化直到 p 等于 NULL 为止，所以最后访问的结点为 *pre)。

Thread(p)算法的思路是：类似于中序遍历的递归算法，在 p 指针不为 NULL 时，先对 *p 结点的左子树线索化，若 *p 结点没有左孩子结点，则将其 lchild 指针线索化为指向其前驱结点的 *pre，将其 ltag 置为 1，若 *pre 结点的 rchild 指针为 NULL，则将其 rchild 指针线索化为指向其后继结点 *p，将其 rtag 置为 1；最后对 *p 结点的右孩子线索化。

中序线索二叉树的算法如下。

```
TBTNode * pre;                                            //全局变量
void  Thread(TBTNode * &p)                                //对二叉树 p 进行中序线索化
{  if(p!=NULL)
   {  Thtead(p→lchild);                                   //左子树线索化
                                                          //此时 * p 结点的左子树不存在或已线索化
      if(p→lchild==NULL)                                  //左孩子不存在:进行前驱结点线索化
       {p→lchild=pre;                                     //建立当前结点的前驱线索
        p→ltag=1;
       }
      else                                                //* p 结点的左子树已经线索化
        p→ltag=0;
      if(pre→child==null)                                 //对 * pre 的后继结点线索化
        {  pre→rchild=p;
           pre→rtag=1;
           }
      else
           pre→rtag=0;
           pre=p;                                         //右子树线索化
           Thread  (p→rchild);
           }
}
TBTNode * CreaThread(TBTNode * &b)                         //中序线索化二叉树
{  TBTNode * root;
   root=(TBTNode *)malloc(sizeof(TBTNode));               //创建头结点
   root→ltag=0;root→rtag=1;
   root→rchild=b;
   if(b==NULL)                                            //空二叉树
     root→lchild=root;
   else
   {  root→lchild=b;
      pre=root;                                           //* pre 是 * p 的前驱结点,供加线索用
      Thread(b);                                          //中序遍历线索化二叉树
      pre→rchild=root;                                    //最后处理,加入指向头结点的线索
      pre→rtag=1;
      root→rchild=pre;                                    //头结点右线索化
   }
return root;
}
```

6.4.3 遍历线索二叉树



视频讲解

遍历线索二叉树,首先要明确先序线索二叉树便于实现先序遍历,中序线索二叉树便于实现中序遍历,后序线索二叉树便于实现后序遍历;其次要注意:在遍历的过程中,两种线索都使用上了吗?还是只使用一种线索即可?

当然,有时在遍历某种次序的线索二叉树时,首先触发开始结点的访问(即先找到该次序下的第一个结点),然后判断该结点是否拥有后继线索,如果存在后继线索,则利用此线索,找到后继结点访问,否则使用相应的遍历法则计算出下一个结点访问,以此类推,直到最

后一个结点。

在先序线索二叉树中查找一个结点的先序后继结点很简单，而查找其先序前驱结点必须知道该结点的双亲结点。同样，在后序线索二叉树中查找一个结点的后序前驱结点也很简单，而查找其后序后继结点也必须知道该结点的双亲结点。由于二叉链中没有存放指向双亲结点的指针，因此在实际应用中，先序线索二叉树和后序线索二叉树较少用到，这里不过多讨论。

在中序线索二叉树中，开始结点就是根结点的最左下结点，而求当前结点在中序序列下的后继结点和前驱结点的方法见表 6.5，最后一个结点的 rchild 指针被线索化为指向头结点。利用这些条件，在中序线索化二叉树中实现中序遍历的算法如下。

```
void  ThInOrder(TBTNode *tb)
{  TBTNode *p=tb->lchild;                      //p 指向根结点
    while(p!=tb)
 {  while(p->ltag==0) p=p->lchild;             //找第一个结点
    printf("%c",p->data);                      //访问第一个结点
    while(p->rtag==1&&p->rchild!=tb)           //使用后继线索访问后继结点
    {   p=p->rchild;
        printf("%c",p->data);
        }
        p=p->rchild;                           //p 无后继线索或者 p 已经到达最后一个结点
    }
}
```

表 6.5　求当前结点在中序序列下的后继结点和前驱结点

求当前结点在中序序列下的后继结点				求当前结点在中序序列下的前驱结点			
p→		rtag		p→		ltag	
		==0	==1			==0	==1
rchild	==root		无后继结点	lchild	==root		无前驱结点
	!=root	后继结点为当前结点右子树的中序下的第一个结点	后继结点为 rchild 指针指示的结点		!=root	前驱结点为当前结点左子树的中序下的最后一个结点	前驱结点为 lchild 指针指示的结点

显然，该算法是非递归的算法，算法的时间复杂度为 O(n)。

6.5　树与森林

6.5.1　树的存储结构

树的存储要求既要存储结点的数据元素本身，又要存储结点之间的逻辑关系。有关树的存储结构有很多，下面介绍 3 种常用的存储结构，即双亲存储结构、孩子链存储结构和孩子兄弟链存储结构。

1. 双亲存储结构

这种存储结构是一种顺序存储结构，用一组连续存储空间存储树的所有结点，同时在每个结点中附设一个指针指示其双亲结点的位置。

双亲存储结构的类型定义如下。

```
typedef struct
{   elemtype data;                          //存放结点的值
    int parent;                             //存放双亲结点的下标
}PTree[MaxSize];
```

例如,图 6.16(a)所示树对应的双亲存储结构如图 6.16(b)所示。其中,根结点 A 的伪指针为−1,其孩子结点 B、C 和 D 的双亲伪指针均为 0,E、F 和 G 的双亲伪指针均为 2。

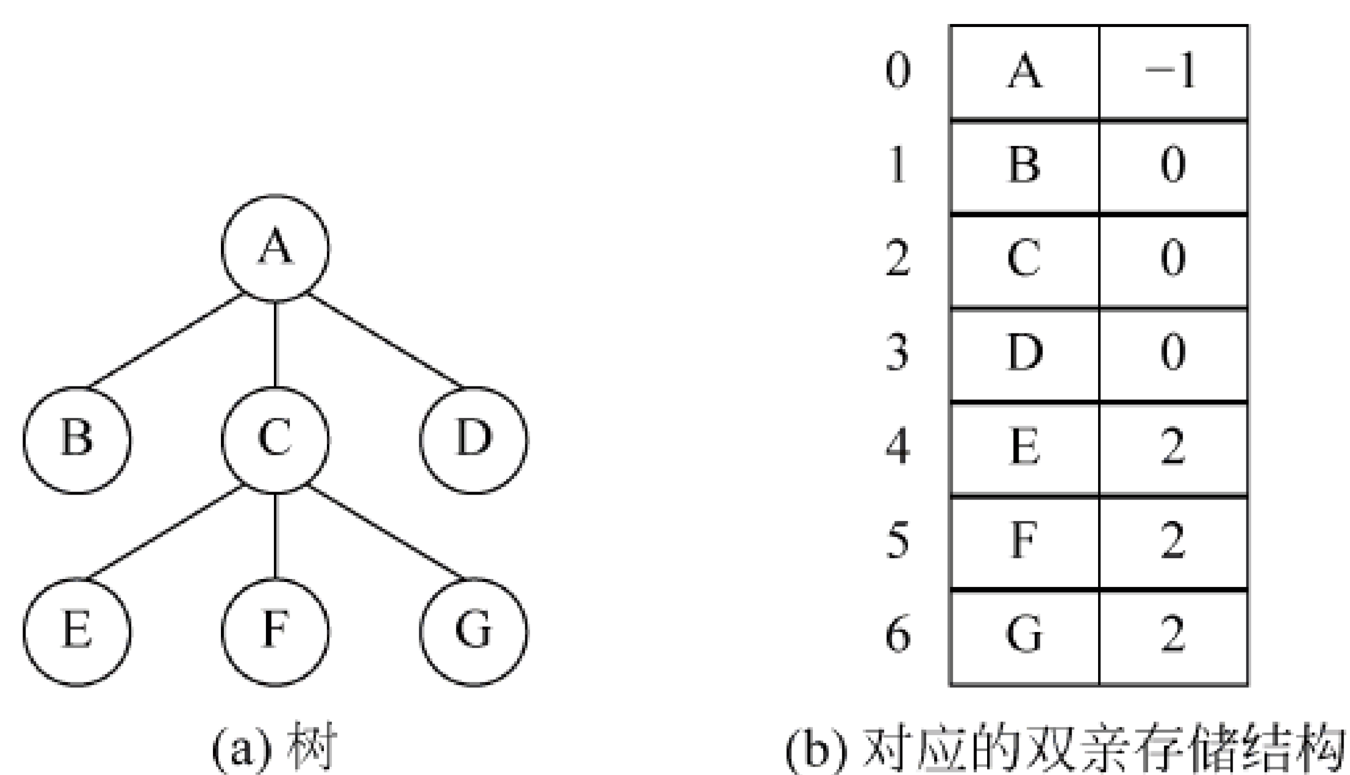


图 6.16 树的双亲存储结构

该存储结构利用了每个结点(根结点除外)只有唯一双亲的性质。在这种存储结构中,求某个结点的双亲结点十分容易,但求某个结点的孩子结点需要遍历整个存储结构。

2. 孩子链存储结构

在这种存储结构中,每个结点不仅包含数据值,还包含指向其所有孩子结点的指针。由于树中每个结点的子树个数(即结点的度)不同,如果按各个结点的度设计成变长结构,则每个结点的孩子结点指针域个数可能各不相同,这就使算法实现非常麻烦。孩子链存储结构按树的度(即树中所有结点度的最大值)设计结点的孩子指针域个数,这样可形成结点的大小一致的定长结构。例如,一棵度为 3 的树,可设置该树中的每个结点均包含 3 个指针域,分别是 child1,child2,child3。

孩子链存储结构的结点类型定义如下。

```
typedef struct node
{ elemtype data;
  struct node *child[Maxdegree];
}TSonNode;
```

其中,Maxdegree 为最多的孩子结点个数,或为该树的度。

例如,图 6.17(a)所示的一棵树,其中树的度为 3,所以在设计其孩子链存储结构时,每个结点指针域个数应为 3,对应的孩子链存储结构如图 6.17(b)所示。

孩子链存储结构的优点是查找某结点的孩子结点十分方便,其缺点是查找某结点的双亲结点比较费时。另外,当树的度较大时。链表存在较多的空指针域,造成资源的浪费;同时,结点只设置了指向孩子结点的指针,没有指向双亲结点的指针,因此想要查找某个结点的双亲时,只能从根结点遍历查找。

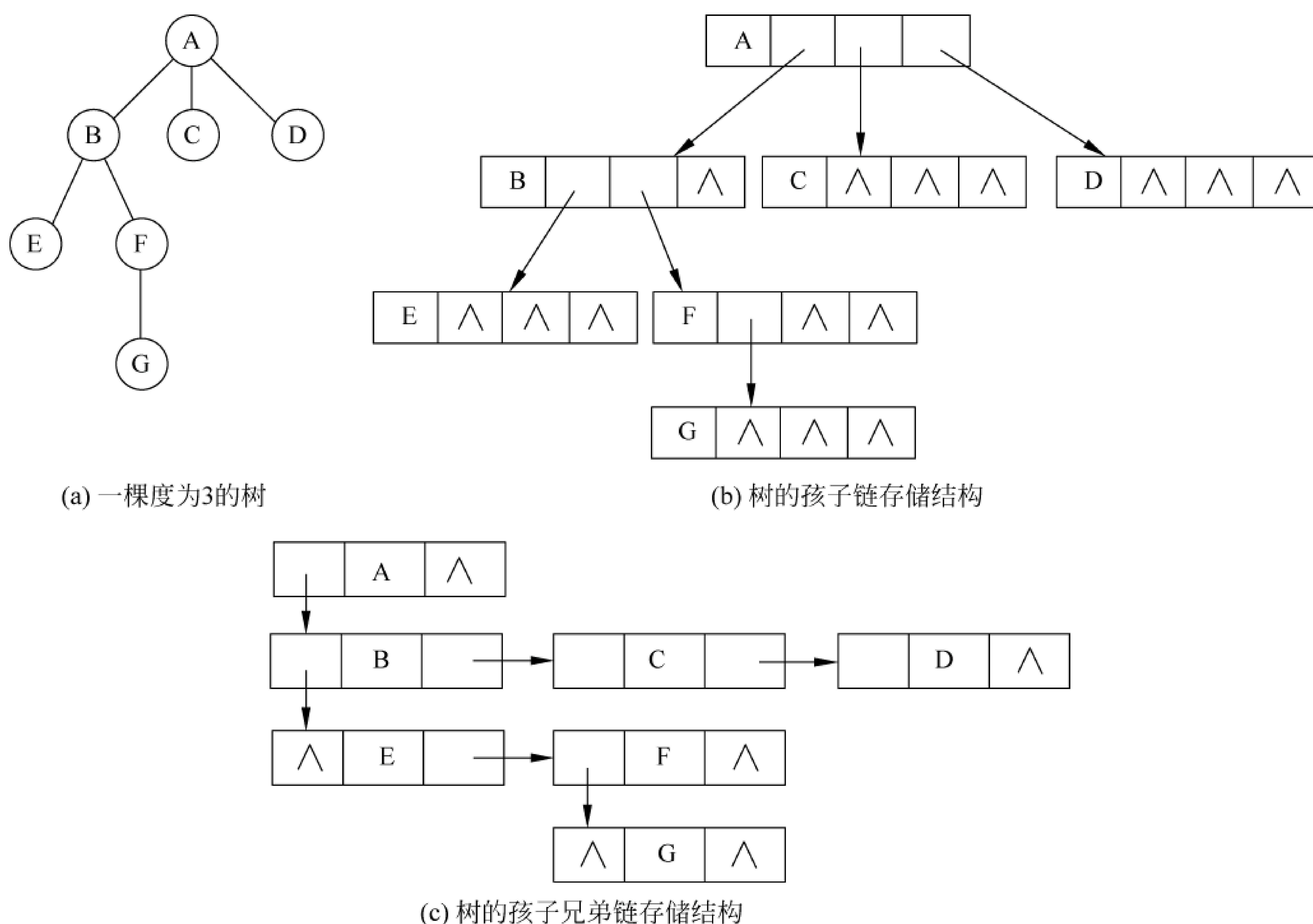


图 6.17　树的链式存储结构

3. 孩子兄弟链存储结构

孩子兄弟链存储是为每个结点设计 3 个域：一个数据元素值域，一个指向该结点的第一个孩子结点的指针域，一个指向该结点的下一个兄弟结点的指针域。

兄弟链存储结构中的结点的类型定义如下。

```
typedef struct tnode
{   elemtype   data;                     //结点的值
    struct tnode   * hp;                 //指向兄弟
    struct tnode   * vp;                 //指向孩子结点
}TSBNode;
```

图 6.17(a)所示的树的孩子兄弟链存储结构如图 6.17(c)所示。

由于树的孩子兄弟链存储结构固定有两个指针域，并且这两个指针域是有序的(即兄弟域和孩子域不能混淆)，所以孩子兄弟链存储结构实际上是把该树转换成为二叉树的存储结构。后面将会讨论，把树转换为二叉树对应的结构恰好就是这种孩子兄弟链存储结构，所以，孩子兄弟链存储结构最大的优点是可方便地实现树和二叉树的互相转换。但是，孩子兄弟链存储结构的缺点也和孩子链存储结构的缺点一样，就是从当前的结点查找其双亲结点比较麻烦，需要从树的根结点开始遍历查找。

【例 6.12】　以孩子兄弟链作为树的存储结构编写一个求树的高度的递归算法。

解：设 f(t)为树 t 的高度，其递归模型为

f(t)=0 若 t==NULL

f(t)=1 若 t 没有孩子结点

$f(t)=\max\limits_{p为t的孩子}(f(p))+1$ 其他情况

递归算法实现如下。

```
int TreeHeight(TSBNode   * t)
{   TSBNode  * p;
    int m, max=0;
    if(t==NULL)
        return(0);                          //空树返回 0
    else if (t->vp==NULL)
        return(1);                          //没有孩子结点返回 1
    else
     {p=t->vp;
      while(p!=NULL)                        //指向第 1 个孩子结点
                                            //从所有孩子结点中找出一个高度最大的孩子
     { m=TreeHeight(p);                     //结点
      if(max<m) max=m;
        p=p->hp;                            //继续求其他兄弟高度
      }
    return(max+1);
    }
}
```

说明：当树采用孩子兄弟链存储结构时，树的算法设计和广义表的算法设计十分相似。

6.5.2 森林与二叉树的转换



视频讲解

树、森林与二叉树之间有一个自然对应的关系，它们之间可以互相转换，即任何一个森林或一棵树都可以唯一对应一棵二叉树，而任一棵二叉树也能唯一对应一个森林或一棵树。正是由于有这样一个一一对应关系，所以可以把在树中处理的问题对应到二叉树中进行处理，从而可以把问题简单化。下面介绍森林、树与二叉树互相转换的方法。

对于一般的树来说，树中结点的左、右次序无关紧要，只要其双亲结点与孩子结点的关系不发生错误就可以了。但在二叉树中，左、右孩子的次序不能随意颠倒。因此，下面讨论的二叉树与一般树之间的转换都约定按照树在图形上的结点次序进行，即把一般树作为有序树处理，这样不至于引起混乱。

1. 森林、树转换为二叉树

若森林 $T=\{T_1,T_2,\cdots,T_m\}$ 是 $m(m\geqslant 0)$ 棵树的序列，则与 T 对应的二叉树 $\beta(T)$ 的构造方法如下。

首先，如果 $m=0$，则 $\beta(T)$ 为空二叉树。

其次，如果 $m>0$，则 $\beta(T)$ 的根结点为 T_1 的根结点，$\beta(T)$ 的根结点的左子树为 $\beta(T_{1,1},T_{1,2},\cdots,T_{1,r})$，其中 $T_{1,1},T_{1,2},\cdots,T_{1,r}$ 是 T_1 的子树；$\beta(T)$ 的根结点的右子树为 $\beta(T_2,T_3,\cdots,T_m)$。

上述方法是一种递归构造方法，如果假定 T 是有序树的序列，那么，由 T 构造出的二叉树 $\beta(T)$ 是唯一的。

用上述递归方法构造的二叉树 β(T)的结点与原来树 T 的结点的关系为：二叉树 β(T)中的任意结点 k，若有左孩子结点，则该左子树结点为 k 原来的最左边(即第一棵)子树的根结点；若有右孩子结点，则该右孩子结点为 k 原来的右边相邻的第一个兄弟结点或右边第一棵相邻树的根结点(当 k 为原森林中树的根结点时)。由此可以把递归构造二叉树 β(T)的过程归纳如下。

step1：在所有相邻兄弟结点(森林中每棵树的根结点可看成兄弟结点)之间加一条水平连线。

step2：对于每个非叶子结点 k，除了其最左边的孩子结点外，删去 k 与其他孩子结点的连线。

step3：所有水平线段以左边结点为轴心顺时针旋转 45°。

通过以上步骤，原来的森林就转换为一棵二叉树。一棵树是森林中的特殊情况，由一棵树转换的二叉树的根结点的右孩子始终为空，原因是一棵树的根结点不存在兄弟结点和相邻的树。

【例 6.13】 将图 6.18(a)所示的森林(由 T_1，T_2，T_3，T_4 4 棵树组成)转换成二叉树。

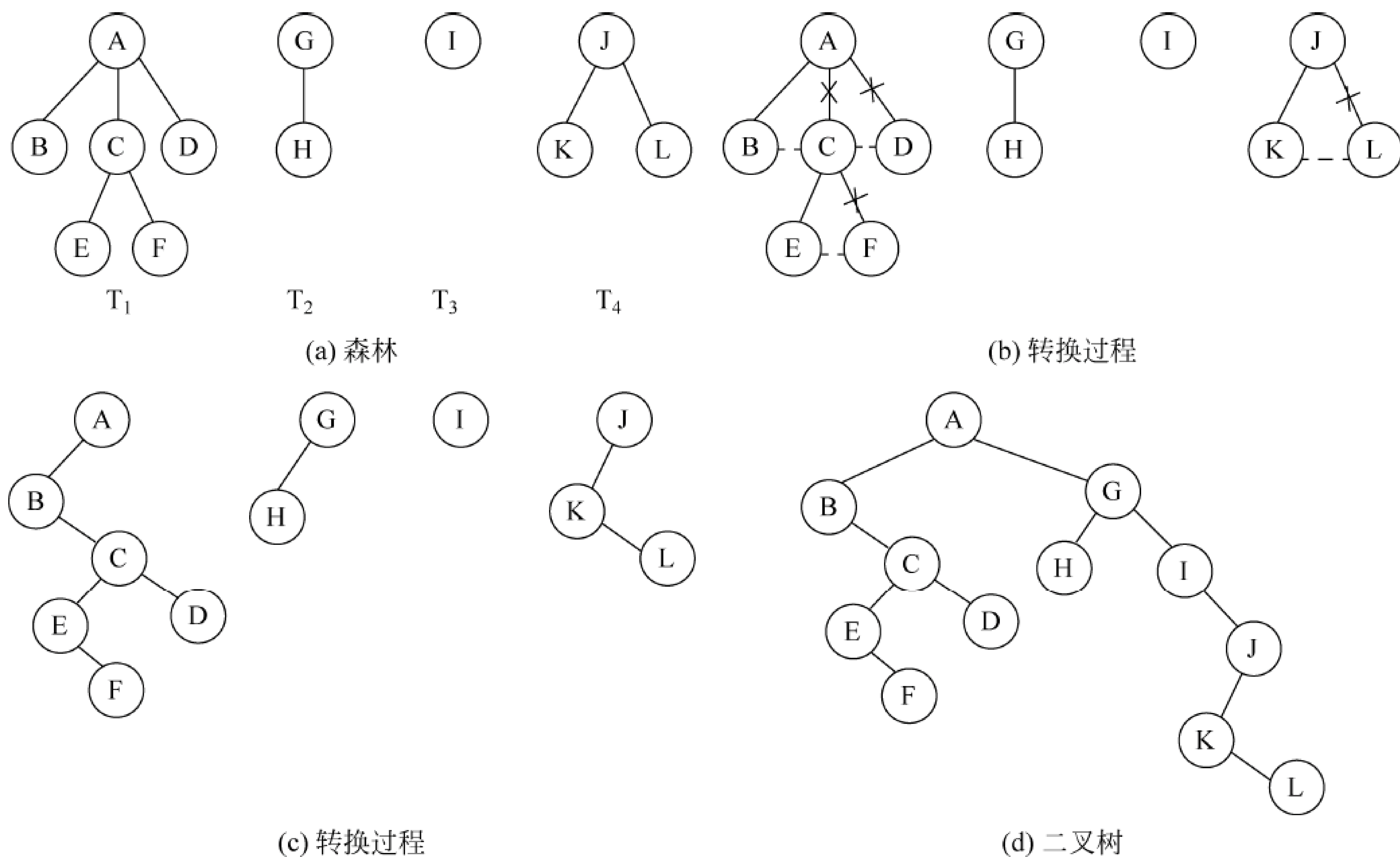


图 6.18 森林转换成二叉树

解：转换为二叉树的过程如图 6.18(b)、(c)所示，最终结果如图 6.18(d)所示。

从森林、树到二叉树的转换过程可以看到以下情况。

step1：原来树中的某结点 a 有多个孩子结点 b_1，b_2，…，b_m时，在二叉树中左孩子结点表示树中 a 结点最左边的孩子结点 b_1，而 b_2 作为 b_1 的右孩子结点，b_3作为 b_2 的右孩子结点，以此类推。也就是说，在转换成的二叉树中，左分支仍表示原来树中的孩子关系，而右分支表示原来树中的兄弟关系。图 6.18(a)中，第一棵树 T_1 中的 A 结点有 B、C、D 3 个孩子结

点,C 结点有 E、F 两个孩子结点,转换成的二叉树中,A 结点有左孩子结点 B,而 B 结点的右孩子为 C,C 结点有右孩子结点 D,C 结点有左孩子结点 E,E 结点的右孩子结点为 F。实际上,树的孩子兄弟链存储结构也是采用这种转换方式,从而得到每个结点只有孩子和兄弟两个指针。

step2:当由 n 棵树的森林转换为二叉树时,除第一棵树外,其余各棵树均变成二叉树中根结点的右孩子树中的结点。图 6.18(a)中的森林有 4 棵树,分别转换成二叉树,结点 A 右链各个二叉树的根结点。

2. 二叉树还原为森林、树

若 β(T)是一棵二叉树,把 β(T)还原为对应的 m(m≥0)棵树序列构成的森林 $T=\{T_1, T_2, \cdots, T_m\}$的方法如下。

首先,如果 β(T)为空二叉树,则 T 为空(m=0)。

其次,如果 β(T)为非空二叉树,则 T 中的根结点为 β(T)的根结点;T_1 中根结点的子树序列$\{T_{1,1}, T_{1,2}, \cdots, T_{1,r}\}$是由 β(T)的左子树还原而成的森林;T 中除 T_1 外的其余树组成的序列$\{T_2, \cdots, T_m\}$是由 β(T)的右子树还原而成的森林。

将由森林或一棵树转换而来二叉树还原为森林或一棵树的过程如下。

step1:对于一棵二叉树中的任一结点 k_1,沿着 k_1 结点的右孩子结点的右子树方向搜索所有右孩子结点,即搜索结点序列 $k_2, k_3, \cdots, k_m$,其中 k_{i+1} 为 k_i 结点($1 \leqslant i < m$)的右孩子结点,k_m 没有右孩子结点。

step2:删去 $k_2, k_3, \cdots, k_m$ 之间的连线。

step3:若 k_1 有双亲结点 k,则连接 k 与 k_i($2 \leqslant i \leqslant m$)。

step4:将图形规整化,使各结点按层次还原为一般的树。

【例 6.14】 将图 6.19 所示的二叉树还原为一般的树。

解:二叉树还原成树的过程如图 6.19 所示。

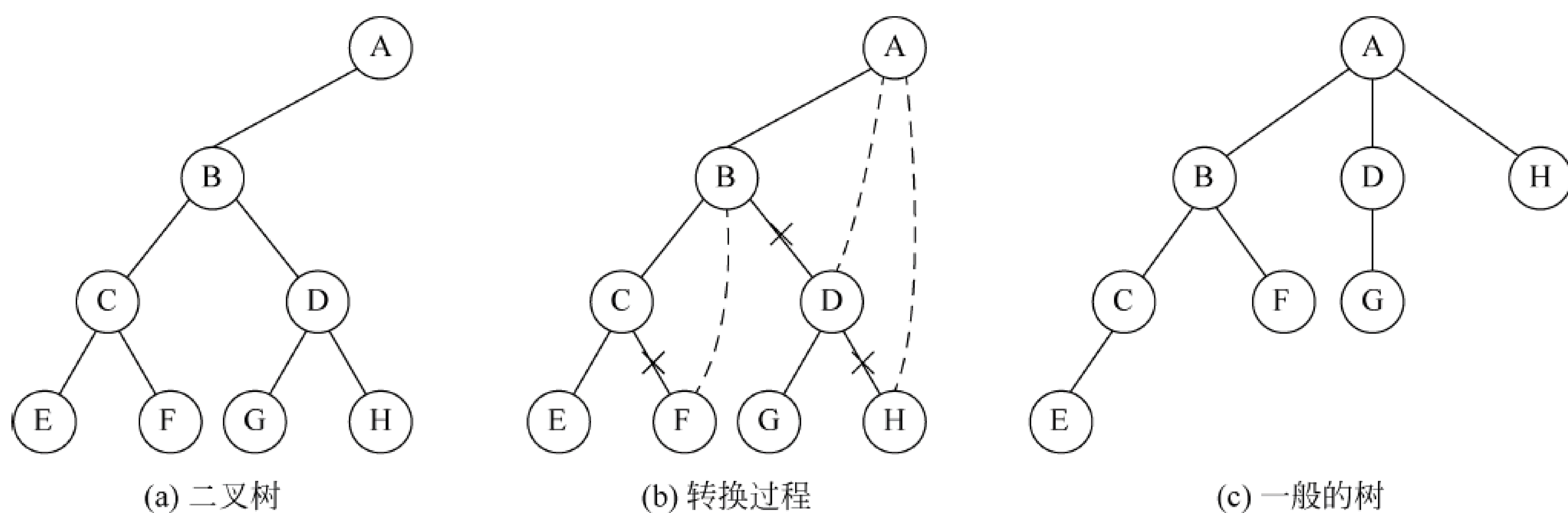


图 6.19 二叉树还原成树的过程

从二叉树到森林、树的还原过程可以看到以下情况。

- 二叉树中某结点的左孩子结点转换为树中该结点的最左边孩子结点,其右孩子结点转换成该结点的兄弟结点。图 6.19(a)中,二叉树 A 结点有左孩子结点 B,而结点 B 有右下孩子结点 D 和 H,还原成森林时,结点 A 有 B、D、H 3 个孩子结点。

- 若二叉树中的根结点有 m 个右下孩子结点，则还原成森林时有 m+1 棵树。图 6.19 中，二叉树的根结点 A 没有右孩子结点（m=0），则还原成森林时只有一棵树。

【例 6.15】 若森林 F 中有 3 棵树 T_1，T_2，T_3，树的结点总数分别为 M 个、N 个、S 个，求森林 F 转换成的二叉树 T 中左子树上结点的个数和右子树上结点的个数。

分析：森林 F 转换为二叉树的过程中，首先将森林中的每棵树分别转换为二叉树（参照左孩子—右兄弟法则），再通过右兄弟法则将转换的后两棵二叉树连接到第一棵二叉树的右子树上，最终合并成一棵二叉树 T。

所以，二叉树 T 左子树上的结点个数=M−1 个，T 右子树上的结点个数=N+S 个。

6.5.3 树的遍历与森林的遍历

由于树是非线性结构，结点之间的关系较线性结构复杂得多，所以树的运算较以前讨论过的各种线性数据结构的算法要复杂许多。

树的运算主要分为 3 大类。

- 寻找满足某种特定关系的结点，如寻找当前结点的双亲结点、孩子结点等。
- 插入或删除某个结点，如在树的当前结点上插入一个新结点或删除当前结点的第 i 个孩子结点等。
- 遍历树中的每个结点。

树的遍历运算是指按某种方式访问树中每一个结点且每一个结点只被访问一次。树的遍历运算主要有先根遍历、后根遍历和层次遍历 3 种。注意，先根遍历和后根遍历算法都可以使用递归实现。图 6.20 所示为一棵树，下面对其进行遍历。

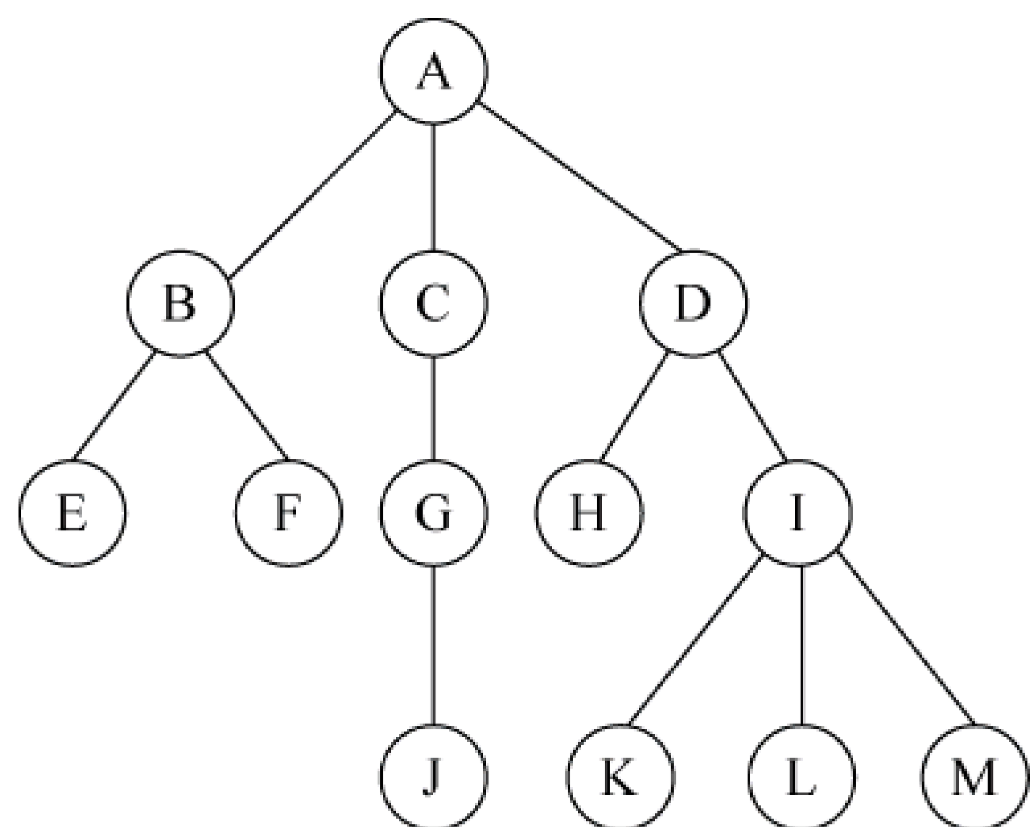


图 6.20 一棵树形结构

1. 先根遍历

先根遍历的过程为

step1：访问根结点。

step2：按照从左到右的次序先根遍历根结点的每一棵子树。

例如，对图 6.20 所示的树，采用先根遍历得到的结点序列为 ABEFCGJDHIKLM。

2. 后根遍历

后根遍历的过程为

step1：按照从左到右的次序后根遍历根结点的每一棵子树。

step2：访问根结点。

例如，对图 6.20 所示的树，采用后根遍历得到的结点序列为 EFBJGCHKLMIDA。

3. 层次遍历

层次遍历的过程为

step1：从根结点开始，从上到下、同一层上从左到右访问树中的每个结点。

例如，对图 6.20 所示的树，采用层次遍历得到的结点序列为 ABCDEFGHIJKLM。

6.6 哈夫曼树及其应用

6.6.1 哈夫曼树的基本概念

在许多应用中，常常将树的结点赋上一个有某种意义的数值，我们称此数值为该结点的**权**。从树根结点到某结点之间的路径长度与该结点权值的乘积称为该结点的带权路径长度。树中所有叶子结点的带权路径长度之和称为该**树的带权路径长度**(**Weight Path Length，WPL**)，通常记为

$$\mathrm{WPL}=\sum_{i=1}^{n} w_i l_i$$

其中，n 表示叶子结点数目；w_i 和 l_i 分别表示叶子结点 k_i 的权值和根到 k_i 之间的路径长度(即从叶子结点到根结点的分支数)。

在 n 个带权叶子结点构成的所有二叉树中，WPL 最小的二叉树称为哈夫曼树(或最优二叉树)。因为构造这种树的算法是由哈夫曼在 1952 年提出的，所以称之为**哈夫曼树**。

例如，给定 4 个叶子结点，设其权值分别为 1，3，5，7，可以构造出 4 棵形状不同的二叉树，如图 6.21 所示。它们的 WPL 分别为

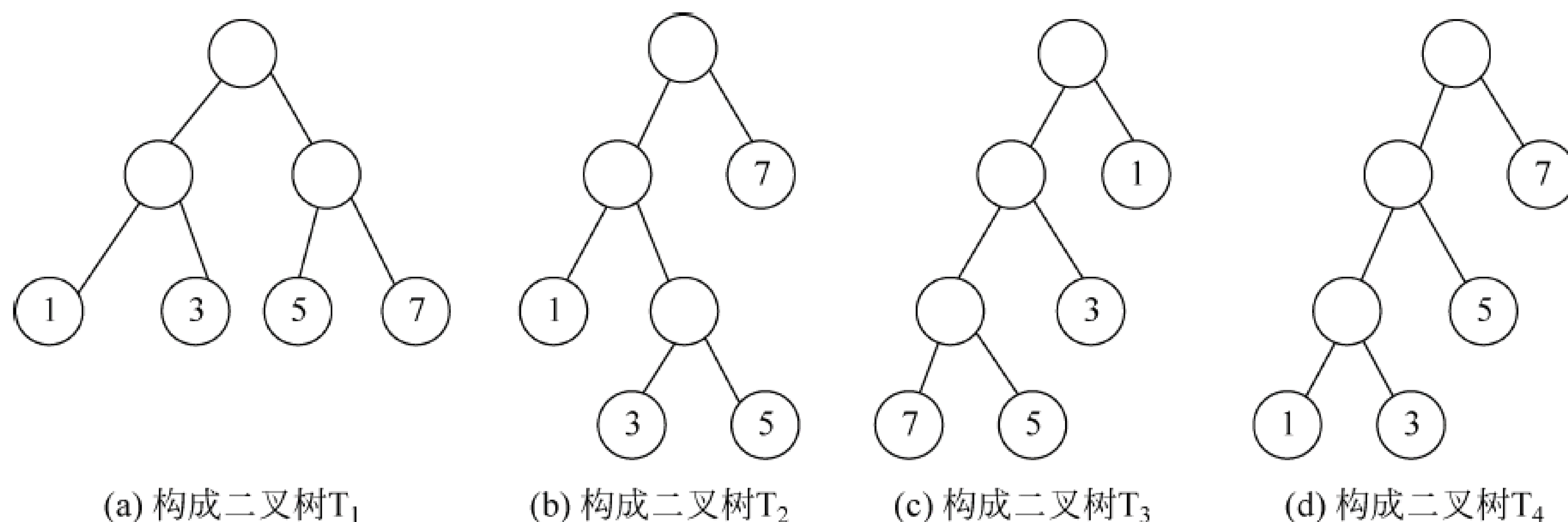


图 6.21　由 4 个叶子结点构成的不同二叉树

T_1 对应的 WPL＝1×2＋3×2＋5×2＋7×2＝32

T_2 对应的 WPL＝1×2＋3×3＋5×3＋7×1＝33

T_3 对应的 WPL＝7×3＋5×3＋3×2＋1×1＝43

T_4 对应的 WPL＝1×3＋3×3＋5×2＋7×1＝29

由此可见，对于一组具有确定权值的叶子结点，可以构造出具有不同 WPL 的二叉树，把其中 WPL 最小的二叉树称为哈夫曼树，又称为最优二叉树。结果表明，图 6.21(d)所示的二叉树是一棵哈夫曼树。观察发现，相同的叶子结点因为分布层次不同导致 WPL 的取值或大或小，权值越大的叶子结点距离根结点越远时 WPL 值偏大，而权值越大的叶子结点距离根结点越近时 WPL 值偏小。



视频讲解

6.6.2 哈夫曼树构造算法

给定 n 个权值，如何构造一棵含有 n 个给定权值的叶子结点的二叉树，使其 WPL 最小呢？哈夫曼最早给出了一个带有一般规律的算法，称为哈夫曼算法。哈夫曼算法如下。

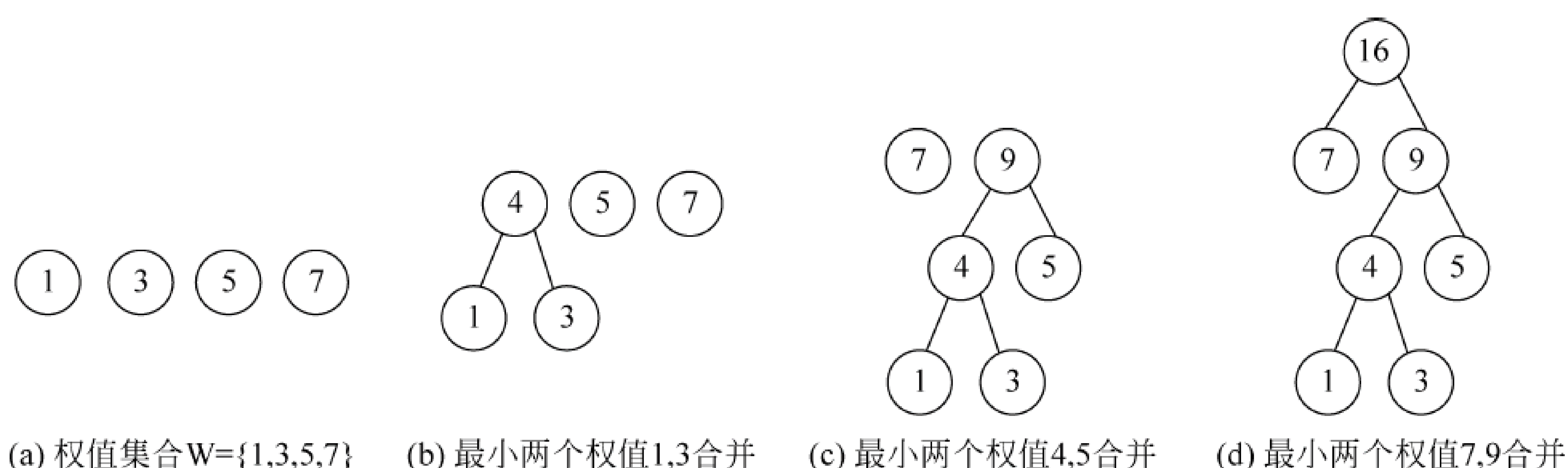
step1：根据给定的 n 个权值{w_1，w_2，…，w_n}，使对应结点构成 n 棵二叉树的森林 T={T_1，T_2，…，T_n}，其中每棵二叉树 T_i(1⩽i⩽n)中都只有一个带权值 w_i 的根结点，其左、右子树均为空。

step2：在森林 T 中选取两棵结点最小的子树分别作为左、右子树构造一棵新的二叉树的森林，且置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和。

step3：在森林 T 中，用新得到的二叉树代替选取的两棵树。

step4：重复 **step2** 与 **step3**，直到 T 只含一棵树停止，这棵树便是哈夫曼树。

例如，假定仍采用给定权值 W={1，3，5，7}构造一棵哈夫曼树，按照上述过程构造哈夫曼树的过程如图 6.22 所示。其中，图 6.22(d)就是最后生成的哈夫曼树，它的 WPL 为 29。



(a) 权值集合W={1,3,5,7}　(b) 最小两个权值1,3合并　(c) 最小两个权值4,5合并　(d) 最小两个权值7,9合并

图 6.22　构造哈夫曼树的过程

定理 6.1　对于具有 n 个叶子结点的哈夫曼树，共有 2n−1 个结点。

证明：在哈夫曼树中不存在度为 1 的结点，即 $n_1=0$，而由二叉树的性质 1 可知 $n_0=n_2+1$，即 $n_2=n_0-1$，$N=n_0+n_1+n_2=n_0+0+n_0-1=2n_0-1$，即 N=2n−1。

为了实现构造哈夫曼树的算法，可设计哈夫曼树中每个结点类型如下。

```
typedef  struct
{
    elemtype  data[5];              //结点值
    int weight;                     //权重
    int parent;                     //双亲结点下标
    int lchild;                     //左孩子结点下标
    int rchild;                     //右孩子结点下标
} HTNode;
```

定义 HTNode ht[ ]数组存放哈夫曼树，对于具有 n 个叶子结点的哈夫曼树，总共有 2n−1 个结点。其算法思路是：先将所有 2n−1 个结点的 parent、lchild 和 rchild 域置为初值−1，处理每个非叶子结点 ht[i](存放在 ht[n]～ht[2n−2]中)：从 ht[0]～ht[i−1]中找出根结点(即其 parent 域为−1)权值最小的两个结点 ht[lnode]和 ht[rnode]，将它们作为 ht[i]的左、右子树，ht[lnode]和 ht[rnode]的双亲结点置为 ht[i]，并且 ht[i].weight=ht[lnode].weight+ht[rnode].weight。如此操作，直到所有 n−1 个非叶子结点都处理完毕。构造哈夫曼树算法如下。

```
void CreateHT(HTNode ht[],int n)
{
    int i,k,lnode,rnode;
    int min1,min2;
    for (i=0;i<2*n-1;i++)                          //所有结点的相关域置初值-1
        ht[i].parent=ht[i].lchild=ht[i].rchild=-1;
    for (i=n;i<2*n-1;i++)                          //构造哈夫曼树
    {
        min1=min2=32767;                           //lnode 和 rnode 为最小权重的两个结点位置
        lnode=rnode=-1;
        for (k=0;k<=i-1;k++)
            if (ht[k].parent==-1)                  //只在尚未构造二叉树的结点中查找
            {
                if (ht[k].weight<min1)
                {
                    min2=min1;rnode=lnode;
                    min1=ht[k].weight;lnode=k;
                }
                else if (ht[k].weight<min2)
                {
                    min2=ht[k].weight;rnode=k;
                }
            }
        ht[i].weight=ht[lnode].weight+ht[rnode].weight;
        ht[i].lchild=lnode;ht[i].rchild=rnode;
        ht[lnode].parent=i;ht[rnode].parent=i;
    }
}
```

显然,构建哈夫曼树算法就是在 ht[ ]中实现树中的每一个结点存储,逻辑上从下向上,直至根结点构建哈夫曼树,反映在 ht[ ]数组是从左向右运算存储树中的每个结点。

6.6.3 哈夫曼编码

在数据通信中,经常需要将传送的文字转换为二进制字符 0 和 1 组成的比特串,这个过程称为编码。为了能有效地进行文字信息的传送,人们设计的编码要保证是前缀编码。所谓**前缀编码**,就是任何字符的编码都不是其他字符编码的前缀,或者任何字符的编码都不能在其他字符编码的前面出现过。同时,我们也希望电文编码的长度最短。哈夫曼树可用于构造使电文编码的总长度最短的编码方案。

具体构造方法如下:设需要编码的字符集合为{d_1,d_2,d_3,…,d_n},各个字符在电文中出现的次数集合为{w_1,w_2,…,w_n},以 d_1,d_2,d_3,…,d_n作为叶子结点,以 w_1,w_2,…,w_n作为各个叶子结点的权值构造一棵哈夫曼树,规定哈夫曼树中的左分支编码为 0,右分支编码为 1,则从根结点到每个叶子结点经过的分支对应的 0 和 1 组成的序列便为该结点对应字符的编码。这样的编码称为**哈夫曼编码**。

哈夫曼编码是一种使用频率越高的字符采用越短的编码。

为了实现构造哈夫曼编码的算法,设计存放每个结点哈夫曼编码的结构类型如下。

```
typedef   struct
{   char cd[N];                                    //存放当前结点的哈夫曼编码
    int start;                                     //cd[start]~cd[n]存放哈夫曼编码
}HCode;
```

由于哈夫曼树中叶子结点的哈夫曼编码长度不同，靠近根结点的编码较短，远离根结点的编码较长，为此采用 HCode 类型变量的 cd[start]～cd[n]存放当前叶子结点哈夫曼编码。对于当前叶子结点 ht[i]，先将对应的哈夫曼编码 hcd[i]的 start 域置初值 n，找其双亲结点 ht[f]。若当前结点是双亲结点的左孩子结点，则在 ht[i]的 cd 数组中添加 0，否则在 ht[i]的 cd 数组中添加 1；然后将 hcd[i]的 start 域减 1。再对其双亲结点进行同样的操作，以此类推，直到无双亲结点(即达到树根结点)为止。最后让 start 指向哈夫曼编码的开始字符。

根据哈夫曼树求对应的哈夫曼编码的算法如下。

```
void CreateHCode(HTNode ht[ ],HCode hcd[ ],int n)
{
    int i,f,c;
    HCode hc;
    for (i=0;i<n;i++)                          //根据哈夫曼树求哈夫曼编码
    {
        hc.start=n;c=i;
        f=ht[i].parent;
        while (f!=-1)                          //循序,直到树根结点
        {
            if (ht[f].lchild==c)               //处理左孩子结点
                    hc.cd[hc.start--]='0';
            else                               //处理右孩子结点
                    hc.cd[hc.start--]='1';
            c=f;f=ht[f].parent;
        }
        hc.start++;                            //start 指向哈夫曼编码最开始字符
        hcd[i]=hc;
    }
}
```

$$\text{哈夫曼编码的总长度} = \sum_{i=1}^{n} d_i \text{ 的编码长度} \times w_i$$

注意：在一组字符的编码中，任一字符的哈夫曼编码不可能是另一字符哈夫曼编码的前缀称为前缀编码。哈夫曼编码是一种最佳的二元前缀码。

【例 6.16】 假定用于通信的电文仅由 a，b，c，d，e，f 这 6 种字符组成，各字母在电文中出现的频率分别是 0.30，0.25，0.20，0.10，0.10，0.05。试为这些字母设计哈夫曼编码。

解：构造哈夫曼树的设计过程如下。

第 1 步：选择频率最低的 f 和 d 构造一棵二叉树，其根结点的频率为 0.15，记为 n_1。

第 2 步：选择频率低的 e 和 n_1 构造一棵二叉树，其根结点的频率为 0.25，记为结点 n_2。

第 3 步：选择频率低的 c 和 b 构造一棵二叉树，其根结点的频率为 0.45，记为结点 n_3。

第 4 步：选择频率低的 n_2 和 a 构造一棵二叉树，其根结点的频率为 0.55，记为结点 n_4。

第 5 步：选择频率低的 n_3 和 n_4 构造一棵二叉树，其根结点的频率为 1.0，记为结点 n_5。

最后构造的哈夫曼树如图 6.23 所示(树中的叶子结点用圆或椭圆表示，分支结点用矩形表示，其中的数字表示结点的频率)，给所有的左分支加上 0，给所有的右分支加上 1，从而得到字符的哈夫曼编码如下。

于是，字符的哈夫曼编码分别为

a:11　　b:01　　c:00　　d:1011
e:100　　f:1010

【例 6.17】 在数据通信中，可以采用 0，1 码的不同排列表示不同的信息。例如，一段报文"CASTCASTSATATATASA"，问：

(1) 传输这段报文采用哈夫曼编码时，编码的总长度 L 等于多少?

(2) 若采用等长编码时，编码的总长度 L 等于多少?

解：(1) 哈夫曼编码是根据哈夫曼树得到的，构建哈夫曼树需要叶子结点的权值作原材料；传输的报文内容中共出现了 C，A，S，T 4 种字符，且它们在报文中出现的次数分别为 2，7，4，5，由此为叶子结点的权值从下向上构建哈夫曼树，如图 6.24 所示。

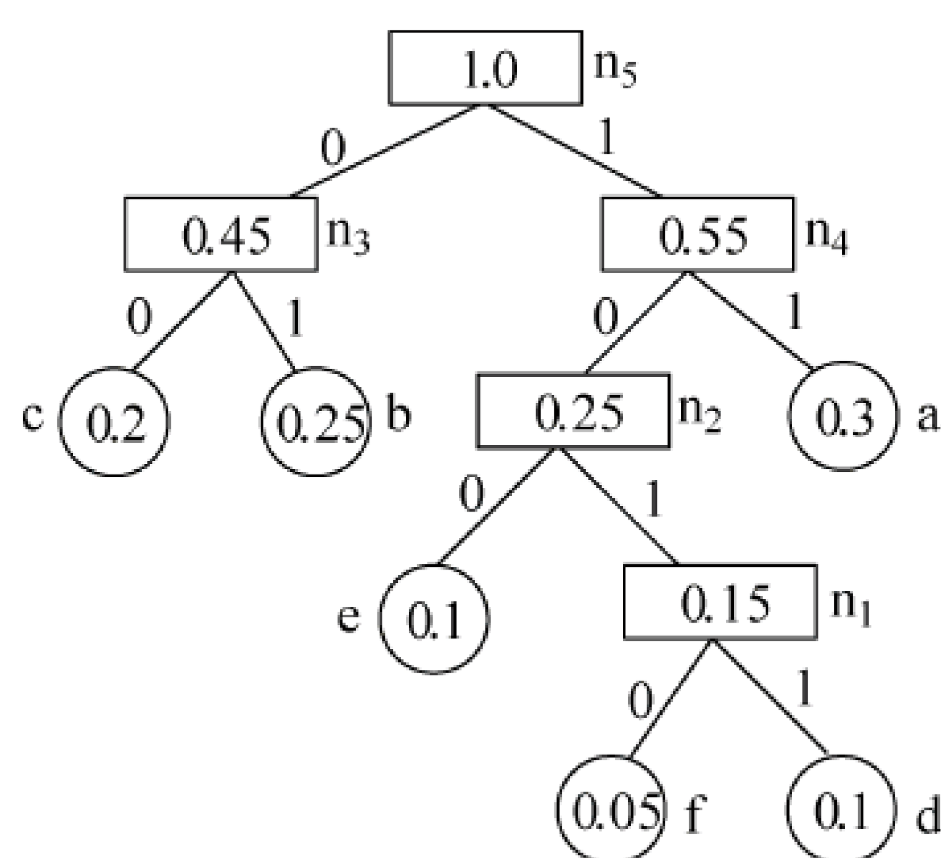


图 6.23　一棵哈夫曼树

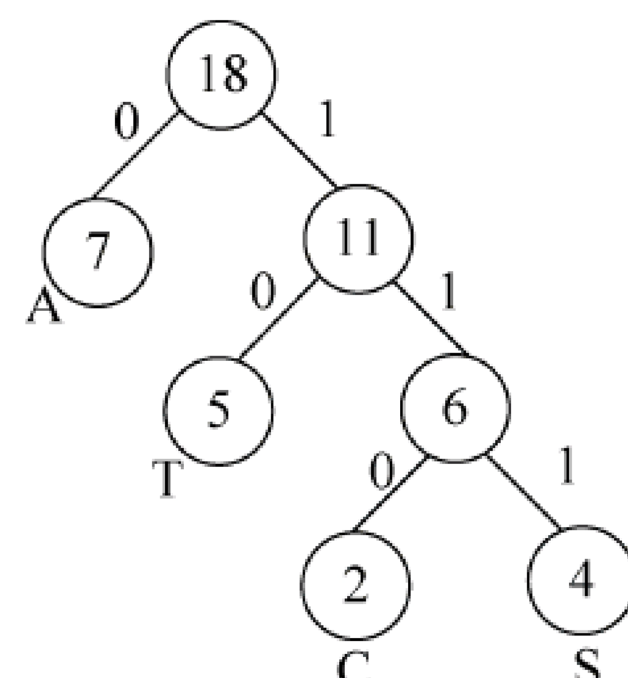


图 6.24　构建的哈夫曼树

于是，字符的哈夫曼编码分别为

C:110　　A:0　　S:111　　T:10

此时编码的总长度 L=2×3+7×1+4×3+5×2=35

(2) 若采用等长编码时，4 种字符可采用长度为 2 的等长编码，如：

C:00　　A:01　　S:10　　T:11

此时编码的总长度 L=2×2+7×2+4×2+5×2=36

注意：哈夫曼编码计算的编码总长度 L 即哈夫曼树的 WPL，而哈夫曼树的 WPL 是最小的，因此哈夫曼编码形成的编码总长度 L 最小。

6.7　STL 中实现树结构

前面曾经介绍过 STL 中的容器，它们几乎与数据结构一一对应。例如，可以使用 list 实现链表，使用 queue 实现队列等。但是很遗憾，STL 中没有 tree 这种容器，但是可以通过 STL 中提供的其他容器间接实现树形结构。

6.7.1　STL 中的 vector

STL 中的 vector 本质上是一个能够存放任意类型的动态数组，可替代 C/C++中的动态数组，其运用起来更具灵活性。

数组在使用上具有一定的不安全性。例如，数组不能进行越界检查，而 vector 在这方面表现良好，因此认为 vector 较数组安全、可靠。另外，数组一旦被定义后，它的容量规模就没法做出改变，这样，对于操作具有相等大的局限性。vector 中的数据可以实现动态添加，即无须预先定义 vector 的容量，可以根据实际需求情况实现动态分配存储空间，这就很好地提高了算法的灵活性。

和 STL 中的其他容器类似，使用 vector 前必须包含其对应的头文件，即通过 include <vector>代码实现。在 vector 容器中有一些常用的函数，如构造函数及其他常用函数。vector 容器的声明方式主要包括以下几种。

```
vector<elem> v                    //创建一个空的 vector
vector<elem> v1(v)                //复制一个 vector
vector<elem> v(n)                 //创建一个 vector,含有 n 个数据且数据均已默认构造产生
vector<elem> v(n, elem)           //创建一个含有 n 个 elem 副本的 vector
vector<elem> v(begin, end)        //创建一个[begin, end)区间的 vector
v.~vector<elem>()                 //销毁所有数据并释放内存空间
```

vector 的实例代码如下所示。

```
#include <iostream>
#include <vector>
using namespace std;
void main() {
  vector<int>::iterator iter;
  vector<int>v1;
  v1 v1.push_back(1);
  v1.push_back(2);
  v1.push_back(3);
  cout << "第一种方式的输出结果:" << endl;
  for(iter = v1.begin(); iter != v1.end(); iter++) {
    cout << *iter << " ";
  }
  cout << endl;
  vector<int> v2(v1);
```

```
    cout << "第二种方式的输出结果:" << endl;
    for(iter = v2.begin(); iter != v2.end(); iter++) {
      cout << *iter << " ";
    }
    cout << endl;
    vector<int> v3(3);
    cout << "第三种方式的输出结果:" << endl;
    for(iter = v3.begin(); iter != v3.end(); iter++) {
      cout << *iter << " ";
    }
    cout << endl;
    vector<int> v4(3, 4);
    cout << "第四种方式的输出结果:" << endl;
    for(iter = v4.begin(); iter != v4.end(); iter++) {
      cout << *iter << " ";
    }
    cout << endl;
    vector<int> v5(v1.begin(), v1.end() - 1);
    cout << "第五种方式的输出结果:" << endl;
    for(iter = v5.begin(); iter != v5.end(); iter++) {
      cout << *iter << " ";
    }
    cout << endl;
    int a[] = {1, 2, 3, 4};
    vector<int> v6(a + 1, a + 2);
    cout << "第六种方式的输出结果:" << endl;
    for(iter = v6.begin(); iter != v6.end(); iter++) {
      cout << *iter << " ";
    }
    cout << endl;
    v6.~vector<int>();
    cout << "第七种方式的输出结果:" << endl;
    for(iter = v6.begin(); iter != v6.end(); iter++) {
      cout << *iter << " ";
    }
    cout << endl;
}
```

vector 中的任意元素或从末尾添加元素都可以在常量级时间复杂度内完成,而查找特定值的元素所处的位置或是在 vector 中插入元素则是线性时间复杂度。常见函数列表如下。

```
增加函数:
void push_back(const T& x):在向量尾部增加一个元素 x
iterator insert(iterator it,const T& x):在向量中迭代器指向元素前增加一个元素 x
iterator insert(iterator it,int n,const T& x):在向量中迭代器指向元素前增加 n 个相同的元素 x
iterator insert(iterator it,const_iterator first,const_iterator last):在向量中迭代器指向元素前插入另
一个相同类型向量的[first,last)间的数据
```

```
删除函数:
iterator erase(iterator it):删除向量中迭代器指向元素
iterator erase(iterator first, iterator last):删除向量中[first, last)间的元素
void pop_back():删除向量中的最后一个元素
void clear():清空向量中的所有元素
遍历函数:
reference at(int pos):返回 pos 位置元素的引用
reference front():返回首元素的引用
reference back():返回尾元素的引用
iterator begin():返回向量头指针,指向第一个元素
iterator end():返回向量尾指针,指向向量最后一个元素的下一个位置
reverse_iterator rbegin():反向迭代器,指向最后一个元素
reverse_iterator rend():反向迭代器,指向第一个元素之前的位置
判断函数:
bool empty() const:判断向量是否为空,若为空,则向量中无元素
大小函数:
int size() const:返回向量中元素的个数
int capacity() const:返回当前向量所能容纳的最大元素值
int max_size() const:返回最大可允许的 vector 元素数量值
其他函数:
void swap(vector&):交换两个同类型向量的数据
void assign(int n, const T& x): 赋 n 个值为 x 的元素到 vector 中,并且清除 vector 中原有的元素
void assign(const_iterator first, const_iterator last):将区间[first, last)的元素设置成当前向量元素
```

此外,vector 有用的方面在于一个 vector 中包含的内容依然可以是 vector 对象。也就是说,vector 允许嵌套声明。这使得 vector 能够很容易地实现像矩阵一样的二维数据结构。如同二维数组一样,但要比数组灵活得多。例如,如下的声明:

```
vector<vector<int> > matrix;
```

需要注意的是,在 C++ 11 之前的标准,为了区别于流操作符号“>>”,在声明嵌套是 vector 对象时,可以在两个尖括号中间添加一个空格,如上所示。

vector 作为 STL 中最常用的容器,基于 vector 对象可以构建一种最简单的树形结构。可使用 vector 容器存储每个结点的孩子结点指针,这种简单的形式可实现一种基于层次化的树形结构。

```
#include <vector>
#include <iostream>
using namespace std;
class Tnode {
  int num;                                    //假设树中共 num 个结点
  vector<Tnode *> * sub;
public:
  Tnode::Tnode(int n) {
    num = n;
```

```
        sub = NULL;
    }
    int getnum() {
        return num;
    }
    vector<Tnode *> * getsub( ) {
        return sub;
    }
    void setnum(int n) {
        this->num = n;
    }
    void setsub(vector<Tnode *> * newsub) {
        this->sub = newsub;
    }
    ...
};

class Tree {
    Tnode * root;
public:
    tree(Tnode * rt) {
        root = rt;
    }
    void displaytree(Tnode * r) {                    //层次遍历树形结构
        cout << r->getnum() << endl;
        if(r->getsub() == NULL) {
            cout << "该结点为叶子结点" << endl;
        return;
        }
        cout << "该结点为分支结点,其子结点为:" << endl;
        for(int i = 0; i < (r->getsub() - size()); i++)
            cout << (r->getsub())->at(i)->getnum() << "";
        cout << endl;
        for(i = 0; i < (r->getsub()) - size(); i++)
            displaytree( (r->getsub())->at(i) );
    }
    ...
}
```

在此并没有完整地将结点类和树类全部实现,可利用 vector 容器的 push_back()方法将每个结点的子树结点集合有效组织在一起,进而形成一棵树,尽管这种实现树的方法十分直观,但实现树的某些运算却比较烦琐。

6.7.2 STL 中的 map

map 是 STL 的一个关联容器,它提供一对一对应关系。map 中存储的每一个数据都是以 key/value 的形式存在的。其中,第一个可以称为关键字,每个关键字只能在 map 中出现一次,第二个可以称为该关键字的值,这种数据处理能力,可以提供编程的快速通道。map

之所以被称为关联性结构，因为它构建了一种从键值到数值之间的映射关系。

例如，一个班级中，每个学生的学号和他的姓名存在一一映射关系，这个模型用 map 可以轻松描述，若学号用 int 描述，姓名用字符串描述，给出 map 的描述代码为

```
map<int,string> mapStudent
```

因为 map 容器中的每个数据都是一个键值——数值对，所以向 map 容器中插入一个元素必须同时提供键值和数值两个数据。map 提供了 insert 方法插入 value_type 数据，insert 方法接口原型：pair<iterator,bool> insert(const value_type& X)，该方法需要构建一个键值对，即 value_type，然后调用 insert 方法，在该方法中实现根据键值对中的 key 值查找对应的结点，如果可查找到，则不插入当前结点，并返回找到的那个结点，将 pair 中的第二个位置为 false；否则插入当前结点，并返回插入的当前结点，且第二个值置为 true。插入结点时，在 map 内部会重新构造一个新的 value_type 结点并将传入的 X 进行 copy 构造，内部使用 placement_new 方式，通过内存分配器分配一个 map 结点，再在获取的结点空间中调用 value_type 构造函数。所以，调用者构造的键值对 value_type 是一个临时变量，不会加到 map 中，这种结点插入的方式是安全的，并判断返回的插入结果，根据插入结果进行后续处理。

```
#include <map>
#include <string>
#include <iostream>
using namespace std;
int main() {
  map<int, string> mapStudent;
  mapStudent.insert(map<int, string>::value_type(1, "one"));
  mapStudent.insert(map<int, string>::value_type(2, "two"));
  mapStudent.insert(map<int, string>::value_type(3, "three"));
  map<int, string>::iterator iter;
  for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++) {
    cout << iter->first << " " << iter->second << endl;
  }
}
```

往 map 里插入数据后，可以使用 size()函数计算当前已经插入了多少数据，用法如下。

```
int nSize=mapStudent.size()
```

STL 中默认是采用小于号排序的(从小到大排序)，若 map 中的关键字是 int 型，如上所示，学生的学号关键字，它本身支持小于号，所以代码在排序上不存在问题。但在一些特殊情况下，如关键字是一个结构体，涉及的排序就会出现问题，因为它没有小于号运算，insert 等函数在编译的时候过不去，这就要求通过小于号重载解决排序问题。

```
#include <map>
#include <string>
```

```
#include <iostream>
using namespace std;
typedef struct tagStudentInfo {
    int nID;
    string strName;
} StudentInfo, * PStudentInfo;                          //学生信息

int main(int argc, char * * argv) {
    map<StudentInfo,int> mapStudent;                     //用学生信息映射分数
    StudentInfo studentInfo;
    studentInfo.nID = 2;
    studentInfo.strName = "one";
    mapStudent.insert(map<StudentInfo,int>::value_type(studentInfo,90));
    studentInfo.nID = 1;
    studentInfo.strName = "two";
    mapStudent.insert(map<StudentInfo,int>::value_type(studentInfo,80));
}
//以上程序无法编译通过,需要重载小于号
typedef struct tagStudentInfo {
    int nID;
    string strName;
    bool operator <(tagStudentInfo const &_A) const {
        //这个函数指定排序策略,按 nID 排序,如果 nID 相等,则按 strName 排序
        if(nID < _A.nID) return true;
        if(nID == _A.nID) return strName.compare(_A.strName) < 0;
        return false;
    }
} StudentInfo, * PStudentInfo;
```

此外,STL 也提供了 3 种方法,对 map 进行遍历:应用前向迭代器、应用反向迭代器、利用数组的形式。如下所示,应用反向迭代器实现 map 遍历。

```
#include <map>
#include <string>
#include <iostream>
using namespace std;
int main() {
    map < int, string >  mapStudent;
    mapStudent.insert(pair < int, string > (1,  "student_one"));
    mapStudent.insert(pair < int, string > (2,  "student_two"));
    mapStudent.insert(pair < int, string > (3,  "student_three"));
    map<int,string >::reverse_iterator iter;
    for(iter = mapStudent.rbegin();  iter  !=  mapStudent.rend(); iter++)
        cout << iter->first << " " << iter->second << endl;
}
```

由于 STL 是一个统一的整体,因此 map 的很多用法都和 STL 中其他的东西结合在一起。此外,map 中由于它内部有序,由红黑树保证,因此很多函数执行的时间复杂度都是

$\log_2 N$,如果用 map 函数可以实现的功能,STL Algorithm 也可以实现,建议用 map 自带函数,这样效率高一些。map 在空间上的特性,由于 map 的每个数据对应红黑树上的一个结点,这个结点在不保存数据时占用 16B,一个父结点指针,左、右孩子指针,还有一个枚举值(标示红黑,相当于平衡二叉树中的平衡因子)。充分利用 map 的强大功能不仅能实现树,甚至还可以实现图。

6.8 综合案例——学校建模问题

1. 问题描述

假设你是一名装修公司的工程师,某天,公司接到一项目任务——为某学院主体教学楼设计装修方案,经理把该项目任务交给了你。图 6.25 所示为该学校主体教学楼的建筑结构俯视图。

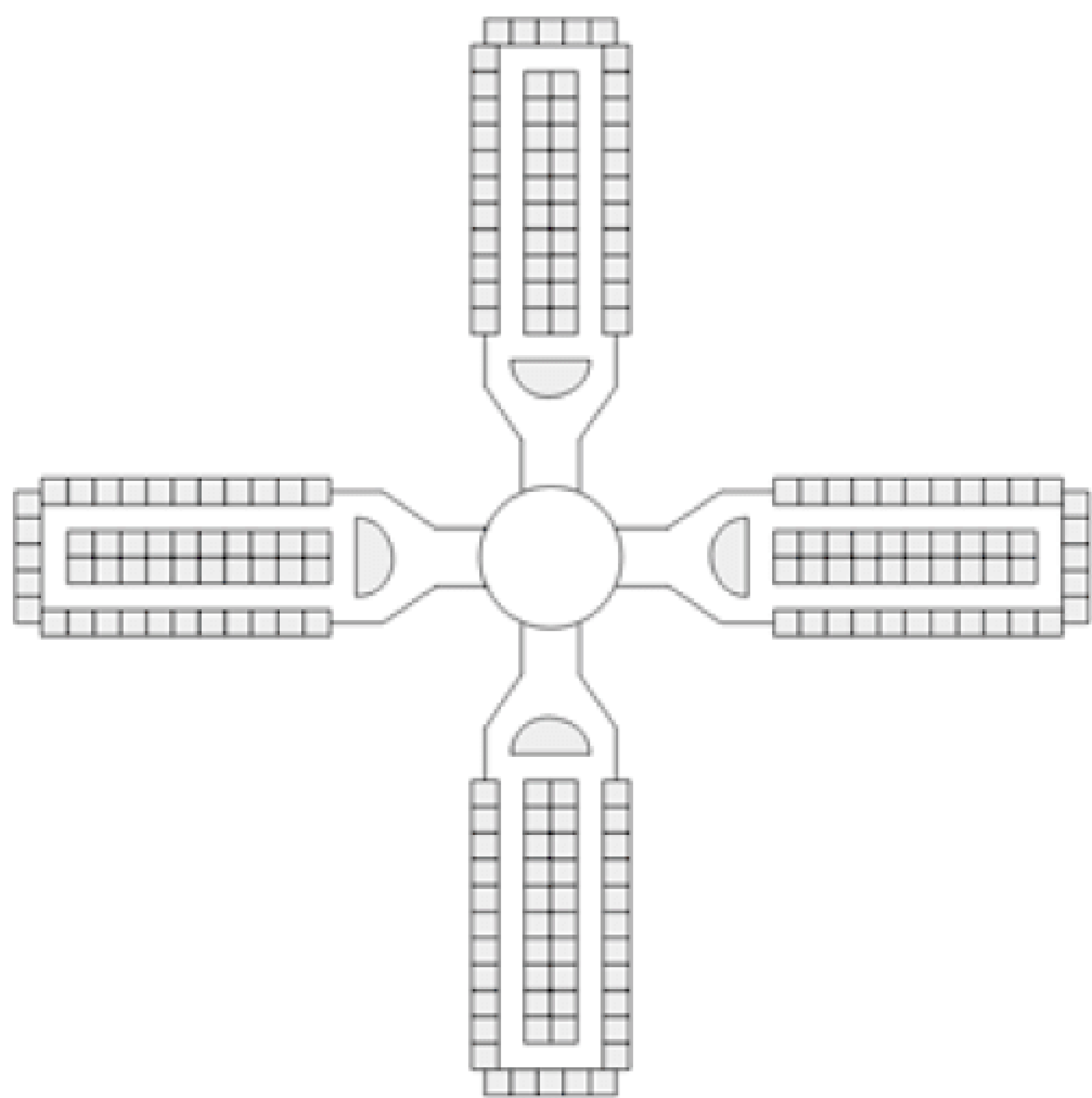


图 6.25 建筑结构俯视图

该教学楼共有 9 个楼层,每个楼层都由 4 个分支及连接分支的中央大厅构成。教学楼的主体由这 4 个分支构成,分支内包含教室、设备实验室、洗手间等。如图 6.26 所示,教学楼的建筑结构被表示成一棵树且树中的每个结点都包含该部分结构的名称及数量。例如,"楼层 9"表示教学楼共有 9 个楼层,"分支 4"表示每层楼都有 4 个分支。

2. 解题思路

现在的任务是编写一个程序,模拟这所学校主体教学楼的建筑结构。所谓模拟教学楼的建筑结构,主要是通过树形结构给出教学楼中每个部分及其子部分之间的关系描述,这需要解决两个问题:一是要求能够正确地得出任意部分的子部分内容。例如,当问及教室由哪些部分组成时,应该得到的结论是每间教室由 100 张课桌、1 张讲桌、1 台多媒体设备和 2 个出口组成;二是程序应当能够统计出在某一部分中另外一个部分的总数。例如,每层楼应该有 48 台多媒体设备。

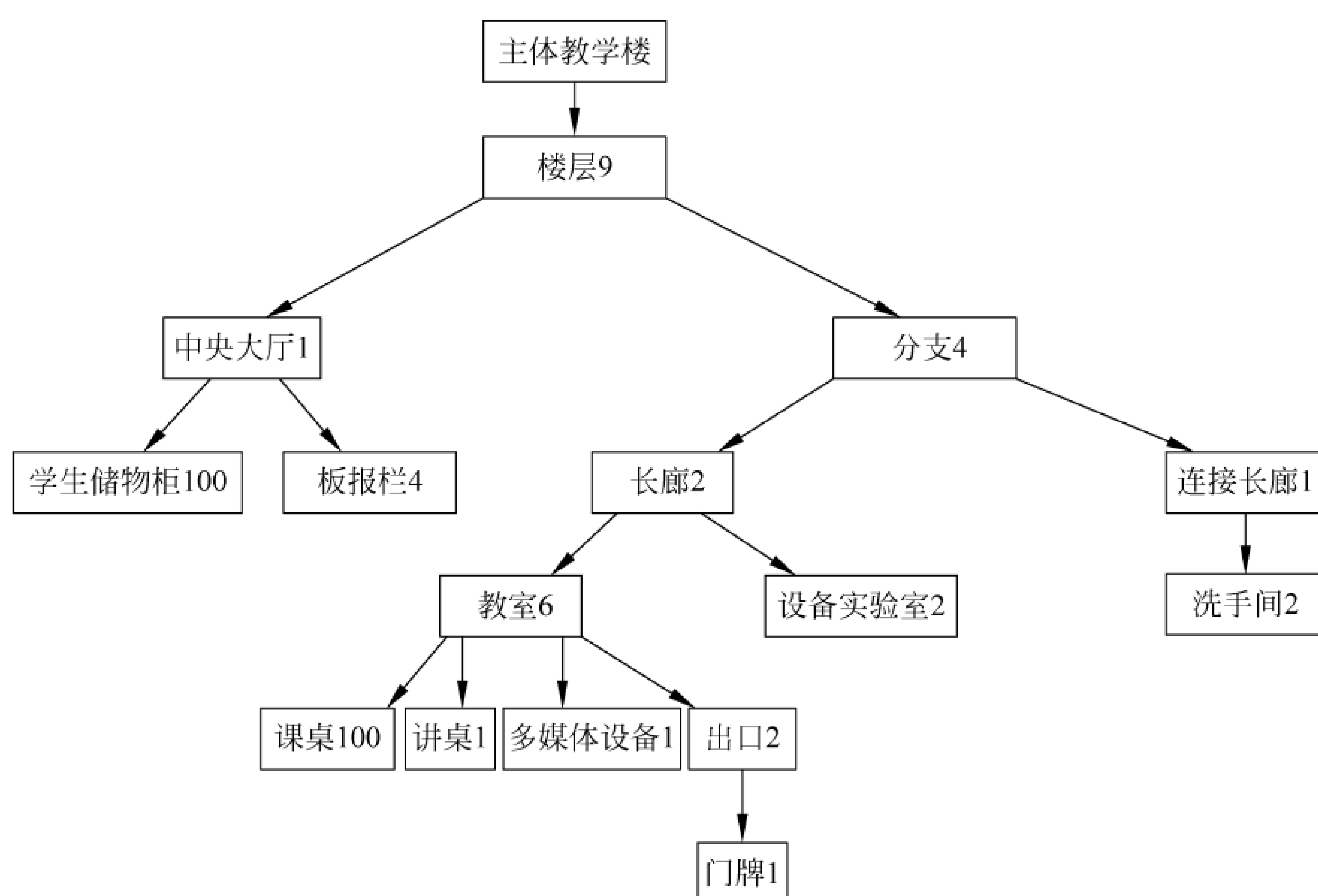


图 6.26 教学楼的建筑结构

3. 代码实现

```
#include <iostream>
#include <fstream>
using namespace std;
#include "parts.h"
void load(char * filename) {
  ifstream inf(filename);
  string part, subpart;
  int quantity;
  while (inf.good()) {
    inf >> part >> quantity >> subpart;
    if (!inf.good()) return;
    add_part(part, quantity, subpart);
  };
}
void whatis(string const &x) {
  Part * xp = partContainer.lookup(x);
  cout << endl;
  xp->describe();
}
void howmany(string const &x, string const &y) {
  Part * xp = partContainer.lookup(x);
  Part * yp = partContainer.lookup(y);
  cout <<"\n"<<y<< " has " << yp ->count_howmany(xp) << " " <<x<< endl;
}
void process(char * filename) {
```

```
    ifstream inf(filename);
    string query, x, y;
    while (inf.good()) {
      inf >> query >> x;
      if (query == "howmany") inf >> y;
      if (!inf.good()) return;
      if (query == "howmany") howmany(x, y);
      else if (query == "whatis")
        whatis(x);
      else {
        cerr << "ERROR!!! Cannot query: " << query << endl;
        return;
      };
    };
}
void main(void) {
    load("definitions.txt");
    process("queries.txt");
}
```

该程序首先载入 definitions. txt 文件,该文本文件给出的是对教学楼中各部分之间关系的描述。

```
hospital 9 floor
floor 1 central_lobby
floor 4 wing
central_lobby 100 locker
central_lobby 4 display_board
wing 2 long_corridor
wing 1 connecting_corridor
long_corridor 6 class_room
long_corridor 2 lab_room
connecting_corridor 2 wash_room
class_room 100 desk
class_room 1 lectern
class_room 1 multi_devices
class_room 2 export
export 1 face_plate
```

接着程序处理"queries. txt"文件。该文件中给出了要求程序调查的内容,其中文件的每行都包括两部分:第一部分是标识符;第二部分是被调查对象。标识符可以有两种,即 whatis 和 howmany。当标识符是 whatis 时,后面的被调查对象是教学楼中某一部分的名称,这时要求程序输出该部分的子部分信息。当标识符是 howmany 时,后面的被调查对象是教学楼中某两部分的名称,这时要求程序输出在后者中前者的总数。例如,howmany multi-devices 要求程序输出教学楼中一共有多少多媒体设备。

下面给出 parts. h 文件的代码清单,该文件包含了此问题中核心问题的求解方法。实现其中的结构和算法时,我们使用了 STL 中的 map 容器。请读者留意程序是如何实现对

教学楼树形结构进行存储的。此外,对树形结构的深入优先搜索也是本题中比较复杂的地方,需要读者仔细思考。

```
#ifndef _PARTS_H_
#define _PARTS_H_
#include <map>
#include <string>
using namespace std;
class Part {
 public:
  string name;
  map<part *, int> subparts;
  Part(string const &n) : name(n) {};
  void describe(void);
  int count_howmany(Part const * p);
  int myCount(map<Part *, int> &myMap, Part const * p, int num);
};
class NameContainer {
 private:
  map<string, Part * >name_map;
 public:
  NameContainer(void) {};
  Part * lookup(string const &name) {
    if (this->name_map.find(name) == this->name_map.end()) {
      Part * part = new Part(name);
      name_map.insert(make_part(name, part));
      return part;
    } else
      return ( * (name_map.find(name))).second;
  }
  //构造函数
  NameContainer() {
    for(map<string, Part * >::iterator it = name_map.begin();
        it != name_map.end(); it++) {
      delete ( * it).second;
    }
  }
};
NameContainer partContainer;
//列出 part 的名字及所有 subparts 与其对应的数量
//并使用游标处理 subparts
void Part::describe(void) {
  cout << "Part " << this->name << " subparts are:" << endl;
  //如果 subparts 不存在,则显示提示信息
  if(subparts.empty()) {
    cout << "There is no subparts!!!" << endl;
    return;
  }
  map<Part *, int>::iterator it = this->subparts.begin();
```

```
    for ( ; it != this->subparts.end(); it++) {
        cout << it->second << " " << it->first->name << endl;
    }
}
//计算 p 所指向部分的实例个数
int Part::count_howmany(Part const * p) {
    return myCount(this->subparts, p, 1);
}
//辅助函数,提供一个数字来记录值
int Part::myCount(map<Part * , int> &myMap, Part const * p, int num) {
    if(myMap.empty()) {
        return 0;
    }
    map<Part * , int>::iterator it;
    for (it = myMap.begin(); it != myMap.end(); it++) {
        num = num * (it->second);
        if(it->first->name == p->name) return num;
        //如果发现结果无效,则回归到原值
        else if(myCount(it->first->subparts, p, num) == 0) {
            num = num / (it->second);
        }
        //进一步搜索
        else
            return myCount(it->first->subparts, p, num);
    }
}
//将命名为 y 的部分添加为 x 部分的子部分
void add_part(string const &x, int q, string const &y) {
    Part * px = partContainer.lookup(x);
    Part * py = partContainer.lookup(y);
    px->subparts.insert(pair<Part * , int>(py, q));
}
#endif
```

本章小结

本章主要介绍了树形结构的基本知识,主要学习要点如下。

- 理解树的定义和基本术语,掌握树的性质及应用。
- 理解二叉树的定义(与度为 2 的树的区别),掌握二叉树的基本性质及应用。
- 掌握满二叉树和完全二叉树的定义和性质。
- 掌握二叉树的顺序存储和二叉链表存储方式。
- 掌握二叉树的基本操作(遍历及其应用)。
- 理解线索二叉树定义和相关概念(如线索、线索化、线索链表等),掌握线索化二叉树过程。
- 理解树、森林与二叉树的转换方式。
- 了解树的存储方式及基本运算。

第4篇　图形结构篇

第7章 图

前面介绍了几种常用的线性结构和树形结构。在树形结构中，结点间具有分支层次关系，每一层上的结点只能与上一层中至多一个结点相关，但可能与下一层的多个结点相关。本章讨论图结构。图结构属于复杂的非线性数据结构，在实际应用中，很多问题可以用图描述。在图结构中，每个元素可以有零个或多个前驱元素，也可以有零个或多个后继元素。也就是说，元素之间的关系是任意的。本章介绍图的基本概念、图的存储结构、图的遍历和相关算法的实现等内容。

7.1 图的概念

7.1.1 图的定义和术语

无论多么复杂的图，都是由顶点和边构成的。采用形式化的定义，**图**(Graph，G)由两个集合V(Vertex)和E(Edge)组成，记为G=(V，E)，其中，V是顶点的有限集合，记为V(G)；E是连接V中两个不同顶点(顶点对)的边的有限集合，记为E(G)。

在G中，如果代表边的顶点对(或序偶)是无序的，则称G为**无向图**。无向图中代表边的无序顶点对通常用圆括号括起来，用以表示一条无向边，如(i，j)表示顶点i和顶点j之间的一条无向边。显然，(i，j)和(j，i)代表的是同一条边。

如果表示边的顶点对(或序偶)是有序的，则称G为**有向图**。有向图中代表边的有序顶点对通常用尖括号括起来，用以表示一条有向边(又称为**弧**)，如<i，j>表示顶点i到顶点j之间的一条有向边，顶点i称为<i，j>的尾，顶点j称为<i，j>的头。通常用由尾指向头的箭头形象地表示一条有向边。可见，<i，j>和<j，i>是两条不同的有向边。

如图7.1所示，从G_1和G_2两图可以看出：两顶点V_1和V_2之间相连的边；无向图G_1中：(V_1，V_2)与(V_2，V_1)相同；有向图G_2中：<V_1，V_2>与<V_2，V_1>不同。

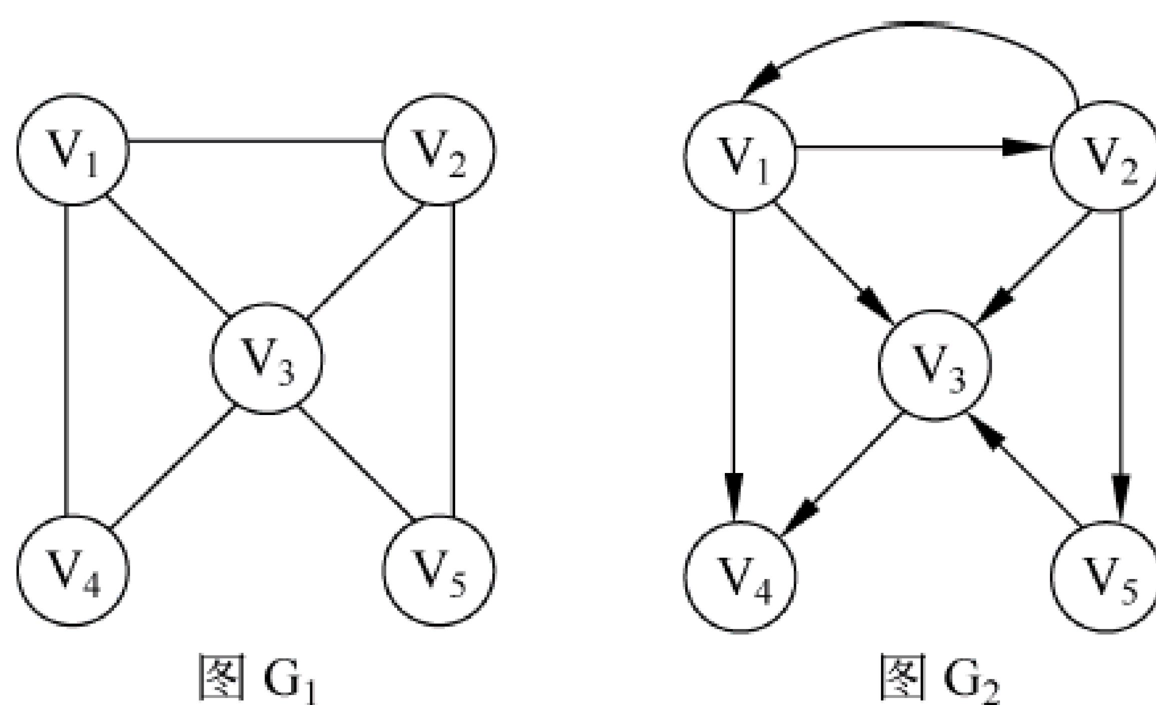


图7.1 无向图G_1和有向图G_2

下面给出图结构中的常用术语。

1. 端点和邻接点

在一个无向图中,若存在一条边(i,j),则称顶点 i 和顶点 j 为该边的两个端点,并称它们互为**邻接点**,即顶点 i 是顶点 j 的一个邻接点,同样,顶点 j 是顶点 i 的一个邻接点。

在一个有向图中,若存在一条边<i,j>,则称此边是顶点 i 的一条**出边**,是顶点 j 的一条**入边**;称顶点 i 和顶点 j 分别为此边的**起始端点**(简称为起点)和**终止端点**(简称终点);称顶点 i 邻接到顶点 j,并称顶点 j 是顶点 i 的**邻接点**,顶点 i 是顶点 j 的**逆邻接点**。

2. 顶点的度、入度和出度

在无向图中,某顶点具有的边的数目称为该顶点的**度**。在有向图中,顶点 i 的度又分为入度和出度,以顶点 i 为终边的入边的数目,称为该顶点的**入度**;以顶点 i 为起点的出边的数目,称为该顶点的**出度**。一个顶点的入度与出度的和为该**顶点的度**。

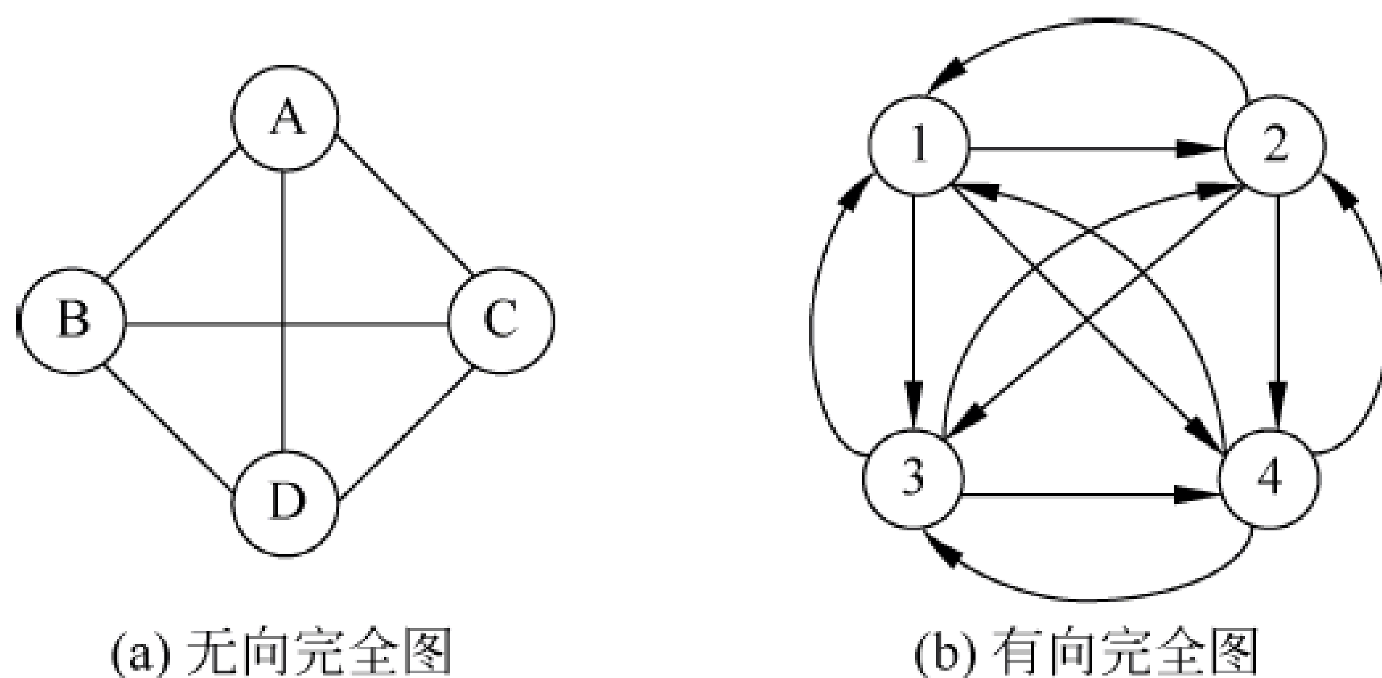
若一个图中有 n 个顶点和 e 条边,每个顶点的度为 $d_i(0 \leqslant i \leqslant n-1)$,则有:

$$e = \frac{1}{2}\sum_{i=0}^{n-1} d_i$$

注意:有向图中所有顶点的入度之和等于所有顶点的出度之和,均等于 e(边数)。

3. 完全图

完全图如图 7.2 所示。



(a) 无向完全图　(b) 有向完全图

图 7.2　完全图

无向完全图:无向图中,任意两顶点都有一直接相连接的边,即有 n(n-1)/2 条边。

有向完全图:有向图中,每两个顶点之间都存在方向相反的两条边,即有 n(n-1)条边。

【例 7.1】 n 个人参加会议,为表示友好,这 n 个人之间相互握手,要求任意两个人都握手且仅握手一次,问这 n 个人共握手几次?

分析:问题是要求计算握手的次数,这些与该 n 个人自身的特点并无关系,因此这 n 个人可抽象为 n 个顶点,每两个人握手且仅握手一次,相当于每两个顶点之间均有边且仅一条边,握手关系是双向关系(相互关系),因此形成的边为无向边,于是 n 个顶点构成了无向完全图。问题就变成了计算无向完全图的边数。

$$(n-1)+(n-2)+\cdots+1+0=n(n-1)/2$$

4. 子图

若一个图 $G_1=(V_1,E_1)$是从 G 中选取部分顶点和部分边(或弧)组成,即 $V_1 \in V, E_1 \in E$,则称 G_1 是 G 的**子图,如图 7.3 所示**。

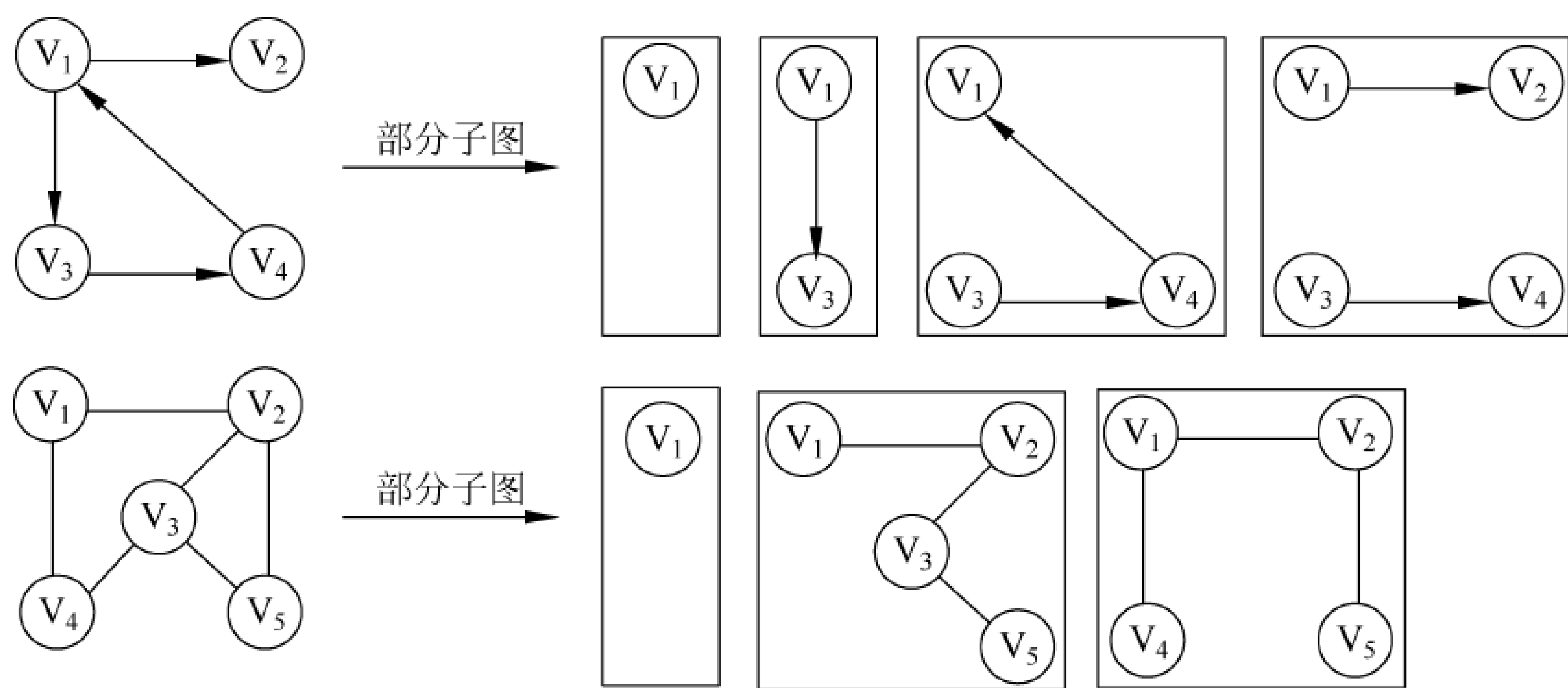


图 7.3 子图

5. 稠密图和稀疏图

如图 7.4 所示，当一个图接近完全图时，称为**稠密图**；相反，当一个图含有较少的边数(即当 e<<n(n-1))时，则称为**稀疏图**。

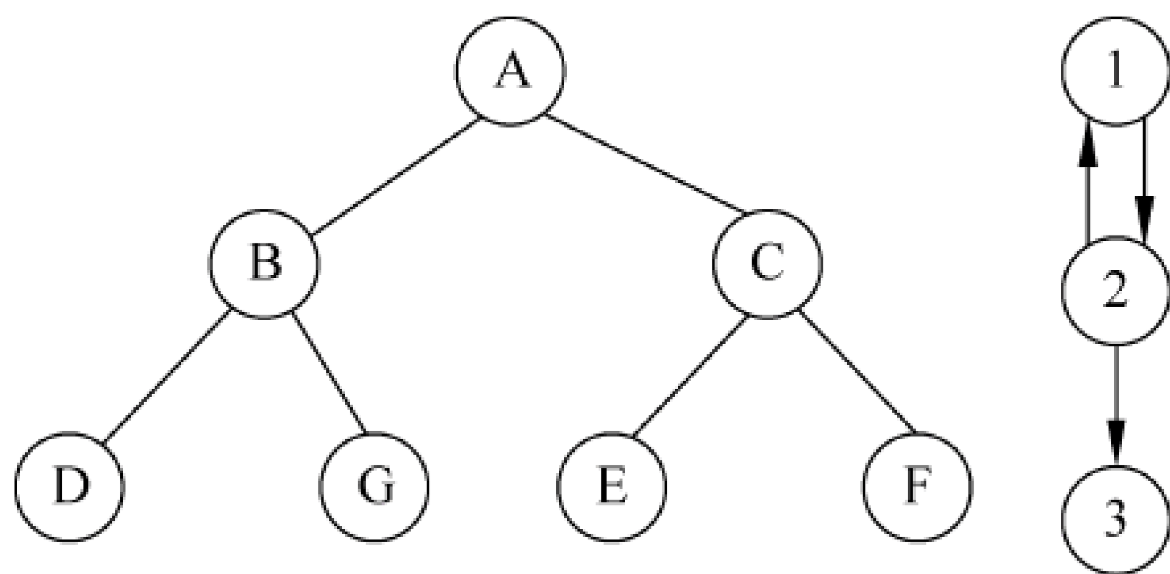


图 7.4 稠密图与稀疏图

6. 路径和路径长度

在一个图 G=(V,E)中，从顶点 i 到顶点 j 的一条**路径**是一个顶点序列$(i,i_1,i_2,\cdots,i_m,j)$，若此图 G 是无向图，则边$(i,i_1),(i_1,i_2),\cdots,(i_{m-1},i_m),(i_m,j)$属于 E(G)；若此图 G 是有向图，则弧$<i,i_1>,<i_1,i_2>,\cdots,<i_m,j>$属于 E(G)。**路径长度**是指一条路径上经过的边或弧的数目。

7. 回路和环

第一个顶点和最后一个顶点相同的路径称为**回路**或**环**，序列中顶点不重复出现的路径称为**简单路径**。除了第一个顶点和最后一个顶点之外，其余顶点不重复出现的回路称为**简单回路**或**简单环**。例如，图 7.2(a)中，(D,B,A,C,D)就是一条简单回路，其长度为 4。

8. 连通、连通图和连通分量

无向图及其连通图如图 7.5 所示。

1）顶点连通

在无向图中，两个不同顶点之间有路径。

2）连通图

在无向图中，任意两个顶点之间都有路径，否则称为**非连通图**。

3）连通分量

无向图中的极大连通子图。

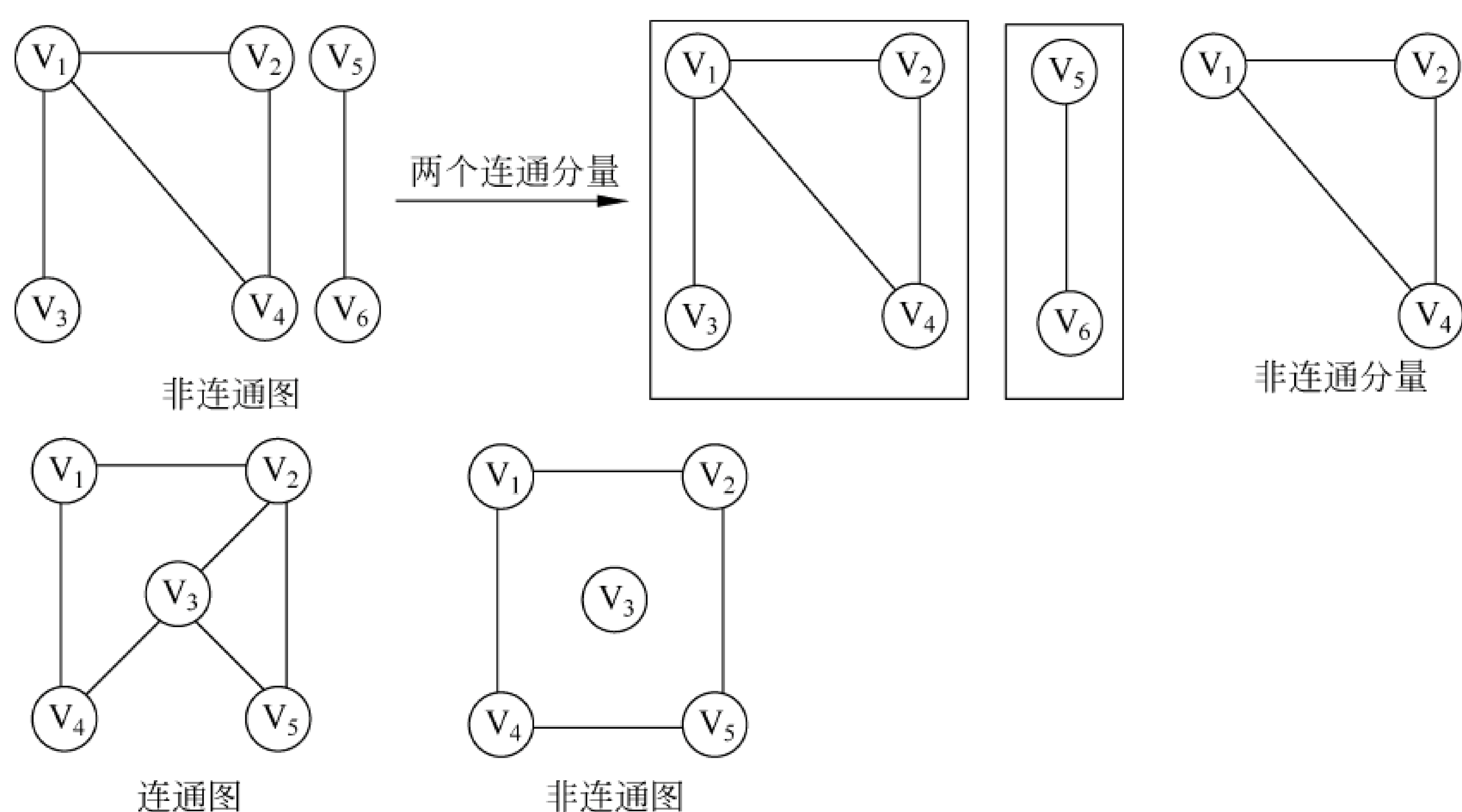


图 7.5　无向图及其连通图

4）极大连通子图

在无向图中，任何不在连通子图中的顶点，加到子图中后，子图就不再是连通的。

注意：

- 顶点间有路径和顶点间有边不同(有边则必定有路径，有路径不一定有边)。
- 连通图的连通分量是其自身，而非连通图可以有多个。

9. 强连通、强连通图和强连通分量

有向图及其连通图如图 7.6 所示。

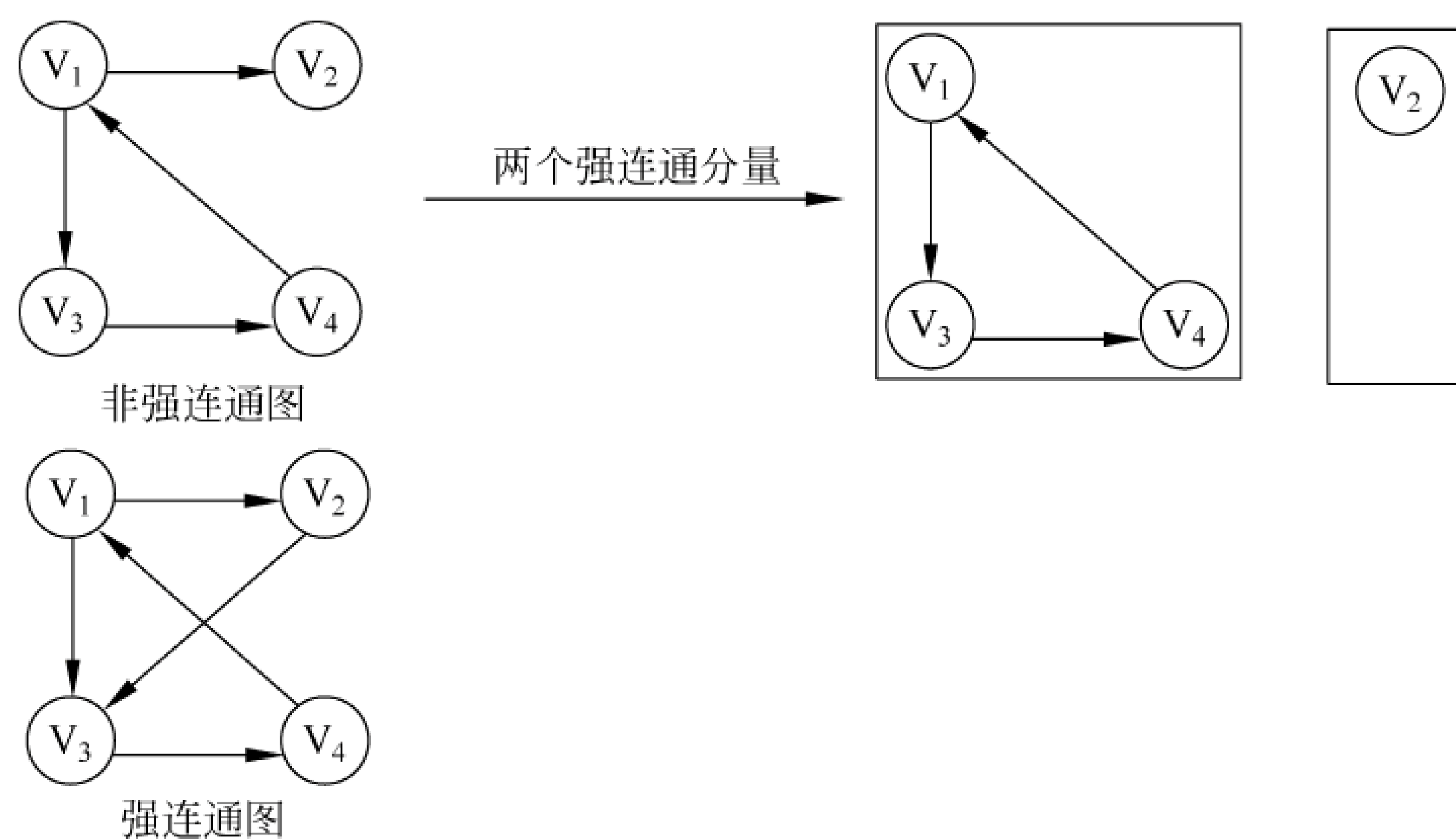


图 7.6　有向图及其连通图

1）顶点连通

在有向图中，两个不同顶点 i,j 之间有路径(顶点 i 到 j 有路径且顶点 j 到顶点 i 也有路径)。

2）强连通图

有向图中，任意两个顶点之间都存在有向路径，即从顶点 i 到顶点 j 和从顶点 j 到顶点 i 都存在路径。否则，其各个强连通子图叫它的**强连通分量**。

强连通图的强连通分量是其自身；而非强连通图的分量可以有多个。

10. 关结点和重连通图

假如在删除图 G 中顶点 i 以及与其相关联的各边后，图的一个连通分量被分割成两个或多个分量，则称顶点 i 为该图的**关结点**。一个没有关结点的连通图称为**重连通图**。

11. 生成树

一个连通图的生成树是一个极小连通子图。

极小连通图：在极小连通子图上，删除任一条边子图就不再连通，若再增加一条边，必定构成一个环。

一个无向图及其生成树如图 7.7 所示。

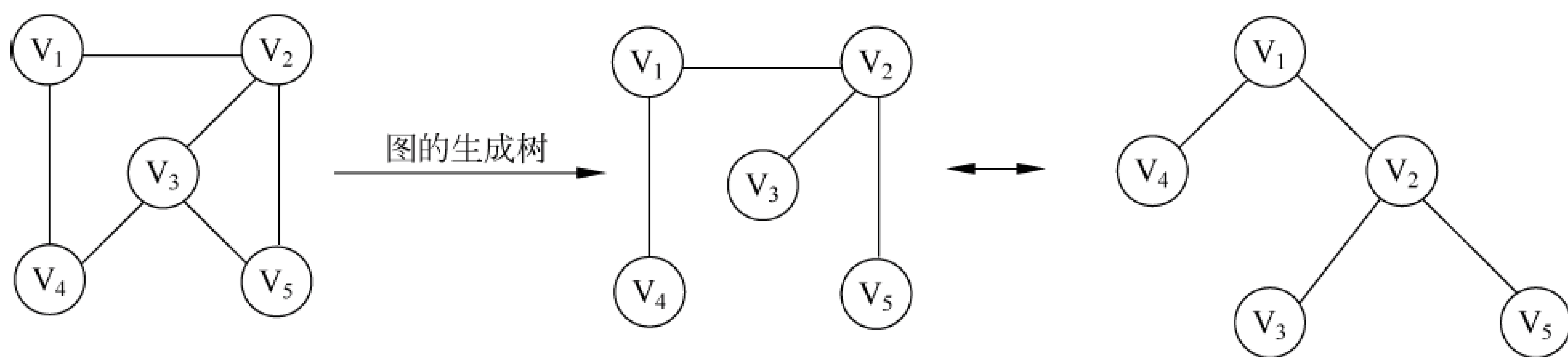


图 7.7　一个无向图及其生成树

生成树的三要素：

- 含有图中所有顶点，即有 n 个顶点。
- 有 n－1 条边。
- 图是连通的。

对非连通图，称由各个连通分量的生成树的集合为此非连通图的**生成森林**。

12. 权和网

图中，每一条边都可以附有一个对应的数值，这种与边相关的数值称为**权**。权可以表示从一个顶点到另一个顶点的距离或花费代价。边上带有权的图称为**带权图**，也称为**网**。例如，图 7.8 是一个带权有向图。

【例 7.2】　n 个顶点的强连通图至少有多少条边？这样的有向图是什么形状？

解：根据强连通图的定义可知，图中的任意两个顶点 i 和 j 都连通，即从顶点 i 到顶点 j 和从顶点 j 到顶点 i 都存在路径。这样，每个顶点的度 $d_i \geqslant 2$，设图中总的边数为 e，则有

$$e = \frac{1}{2}\sum_{i=0}^{n-1} d_i \geqslant \frac{1}{2}\sum_{i=0}^{n-1} 2 = n$$

即 $e \geqslant n$。因此，n 个顶点的强连通图至少有 n 条边。

刚好只有 n 条边的强连通图是环形的，即顶点 0 到顶点 1 有一条有向边…顶点 n－1 到顶点 0 有一条有向边，如图 7.9 所示。

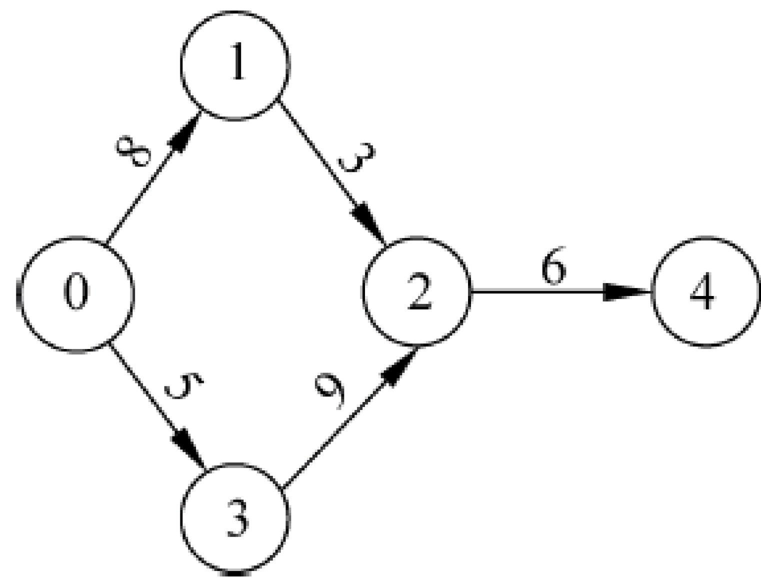


图 7.8　带权有向图

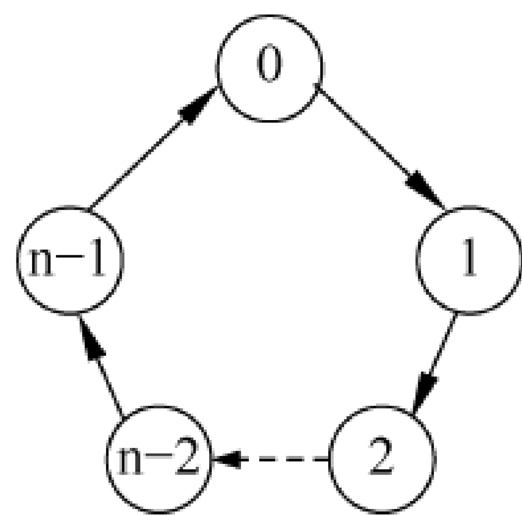


图 7.9　具有 n 个顶点 n 条边的强连通图

7.1.2 图的抽象数据类型

图是一种数据结构,加上一组基本操作,就构成了抽象数据类型。

图的抽象数据类型定义如下。

ADT Graph
{
数据对象:D={具有相同特性的数据元素的集合,称为顶点集}。
数据关系:R={r}
r={<v,w>|v,∈ V且P(v,w),<v,w>表示从v到w的弧,谓词P(v,w)定义了弧<v,w>的意义或信息 ,其中w个元素可以有零个或多个前驱元素,可以有零个或多个后继元素}
基本操作:
CreateGraph(&G,V,E) :创建图G。
　　初始条件:V是图的顶点集,E是图中弧的集合。
　　操作结果:按V和E的定义构造图G。
DestroyGraph(&G) :销毁图G。
　　初始条件:图G存在。
　　操作结果:销毁图G。
LocateVex(G,u) :查找顶点在图中的位置。
　　初始条件:图G存在,u和G中的顶点有相同特征。
　　操作结果:若G中存在顶点u,则返回该顶点在图中的位置;否则返回其他信息。
GetVex(G,v) :获取图中某个顶点的值。
　　初始条件:图G存在,v是G中的某个顶点。
　　操作结果:返回v的值。
PutVex(&G,v,value) :为图中的某个顶点赋值。
　　初始条件:图G存在,v是G中的某个顶点。
　　操作结果:对v赋值value。
DFS(G,Visit()) :图的深度优先遍历。
　　初始条件:图G存在,Visit是顶点的应用函数。
　　操作结果:对图进行深度优先遍历。在遍历过程中对每个顶点调用函数Visit一次且仅一次,一旦Visit()失败,则操作失败。
BFS(G,Visit()) :图的广度优先遍历。
　　初始条件:图G存在,Visit是顶点的应用函数。
　　操作结果:对图进行广度优先遍历。在遍历过程中对每个顶点调用函数Visit一次且仅一次,一旦Visit()失败,则操作失败。
}**ADT** Graph

通常用字母或自然数(顶点的编号)识别图中的顶点。约定用i(0≤i≤n-1)表示第i个顶点的编号。E(G)表示图G中边的集合,它确定了图G中的数据元素之间的关系。E(G)可以为空集,当E(G)为空集时,图G只有顶点,而没有边。

7.2 图的存储表示

图的结构比较复杂,任意两顶点之间都可能存在关系,因此无法以数据元素在存储区中的物理位置表示元素之间的关系,即图难以采用顺序映像的存储结构,和前面介绍的所有数据结构不同,图的存储结构至少要保存:

- 顶点本身的信息。

- 各个顶点之间的关系。



7.2.1 邻接矩阵

借助数组的数据类型表示元素之间的关系的方法是邻接矩阵法。邻接矩阵是表示顶点之间相邻关系的矩阵,即邻接矩阵是表示边的矩阵。设 G=(V,E)是含有 n(n>0)个顶点的图,n 个顶点的编号为 0～(n−1),则 G 的邻接矩阵 A 是 n 阶方阵,其定义如下。

(1) G 是不带权的无向图,则

$$A[i][j]=\begin{cases}1 & 若(i,j)\in E(G)\\ 0 & 其他\end{cases}$$

(2) G 是带权的无向图,则

$$A[i][j]=\begin{cases}W_{ij} & 若\ i\neq j\ 且(i,j)\in E(G)\\ 0 & i=j\\ \infty & 其他\end{cases}$$

(3) G 是不带权的有向图,则

$$A[i][j]=\begin{cases}1 & 若<i,j>\in E(G)\\ 0 & 其他\end{cases}$$

(4) G 是带权的有向图,则

$$A[i][j]=\begin{cases}W_{ij} & 若\ i\neq j\ 且<i,j>\in E(G)\\ 0 & i=j\\ \infty & 其他\end{cases}$$

例如,图 7.1 的无向图 G_1、有向图 G_2 和图 7.8 中的带权有向图分别对应的邻接矩阵 A_1、A_2 和 A_3 如图 7.10 所示。

$$A_1=\begin{bmatrix}0&1&1&1&0\\1&0&1&0&1\\1&1&0&1&1\\1&0&1&0&0\\0&1&1&0&0\end{bmatrix}\quad A_2=\begin{bmatrix}0&1&1&1&0\\1&0&1&0&1\\0&0&0&1&0\\0&0&0&0&0\\0&0&1&0&0\end{bmatrix}\quad A_3=\begin{bmatrix}0&8&\infty&5&\infty\\\infty&0&3&\infty&\infty\\\infty&\infty&0&\infty&6\\\infty&9&\infty&0&\infty\\\infty&\infty&\infty&\infty&0\end{bmatrix}$$

图 7.10 图的邻接矩阵

注意:邻接矩阵的特点如下。

- 图的邻接矩阵表示是唯一的。
- 对于含有 n 个顶点的图,采用邻接矩阵存储时,无论是有向图,还是无向图,也无论边的数目是多少,其存储空间均为 $O(n^2)$,所以邻接矩阵适合于存储边的数目较多的稠密图。
- 便于计算顶点的度如下。
  - 无向图的度:顶点 V_i 的度为第 i 行(或第 i 列)的非零元素个数。
  - 有向图的度:第 i 行的非零元素(或非∞元素)的个数是顶点 V_i 的出度;第 j 列的非零元素(或非∞元素)的个数是顶点 V_j 的入度。
- 无向图的邻接矩阵一定是对称矩阵。因此,可以按照压缩存储的思想,在具体存放邻接矩阵时只存放下(或上)三角矩阵的元素即可。有向图的邻接矩阵不一定是对

称矩阵。

- 用邻接矩阵方法存储图,很容易确定图中任意两点之间是否有边相连。但是,要确定图中有多少条边,则必须按行、按列对每个元素进行检测,花费的时间代价很大,这是用邻接矩阵存储图的局限性。

邻接矩阵的数据类型定义如下。

```
#define MAXV<最大顶点的个数>
typedef struct VertexType
{ int no;                          //顶点编号
  InfoType info;                   //顶点其他信息
}VertexType;                       //顶点类型
typedef struct MGraph
{ int edges[MAXV][MAXV];           //邻接矩阵的边数组
  int n,e;                         //顶点数,边数
  VertexType vexs[MAXV];           //存放顶点信息
}MGraph;                           //完整的图邻接矩阵类型
```

7.2.2 邻接表



视频讲解

图的邻接表存储方法是一种顺序分配与链式分配相结合的存储方法。在表示含 n 个顶点的图的邻接表中,为每个顶点建立一个单链表,第 i(0≤i≤n−1)个单链表中的结点表示依附于顶点 i 的边(对有向图是以顶点 i 为尾的边)。每个单链表上附设一个表头结点,将所有表头结点构成一个表头结点数组。边结点(或表结点)和表头结点的结构如下。

边结点(表结点)

adjvex	nextarc	info

表头结点(顶点结点)

data	firstarc

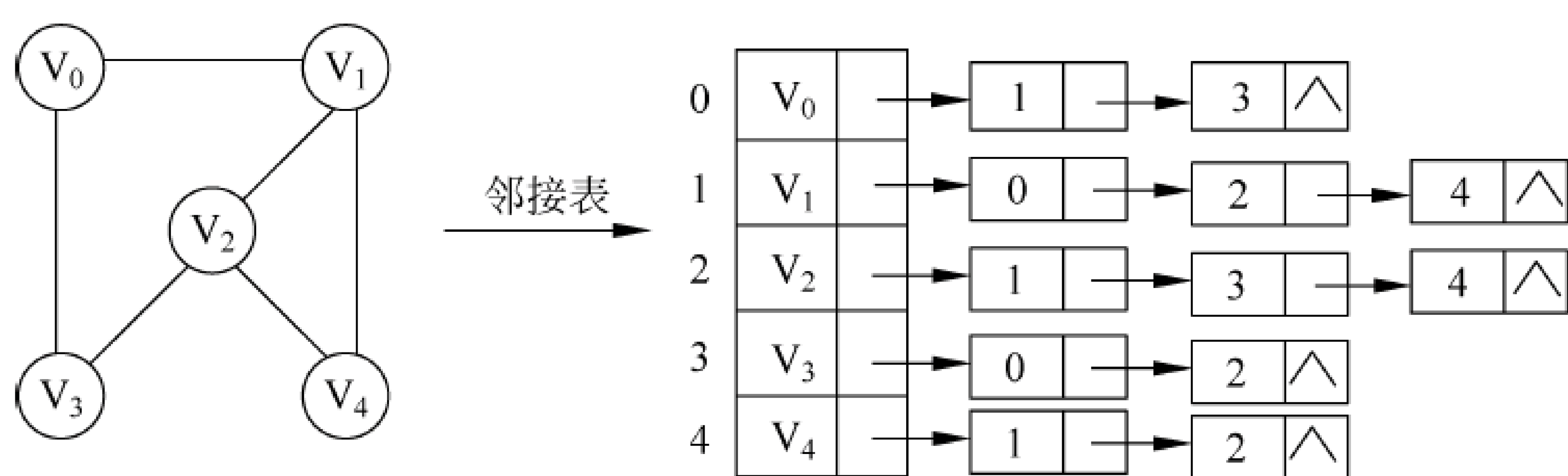
其中,边结点由 3 个域组成,adjvex 指示与顶点 i 相邻的顶点的编号(即邻接点的下标),nextarc 指向对应下一条边的结点,info 存储与边相关的信息,如权值等。表头结点由两个域组成,data 存储顶点 i 对应的名称或其他信息,firstarc 指向对应顶点 i 的链表中的第一个边结点。

例如,无向图的邻接表如图 7.11(a)所示。有向图的邻接表如图 7.11(b)所示。

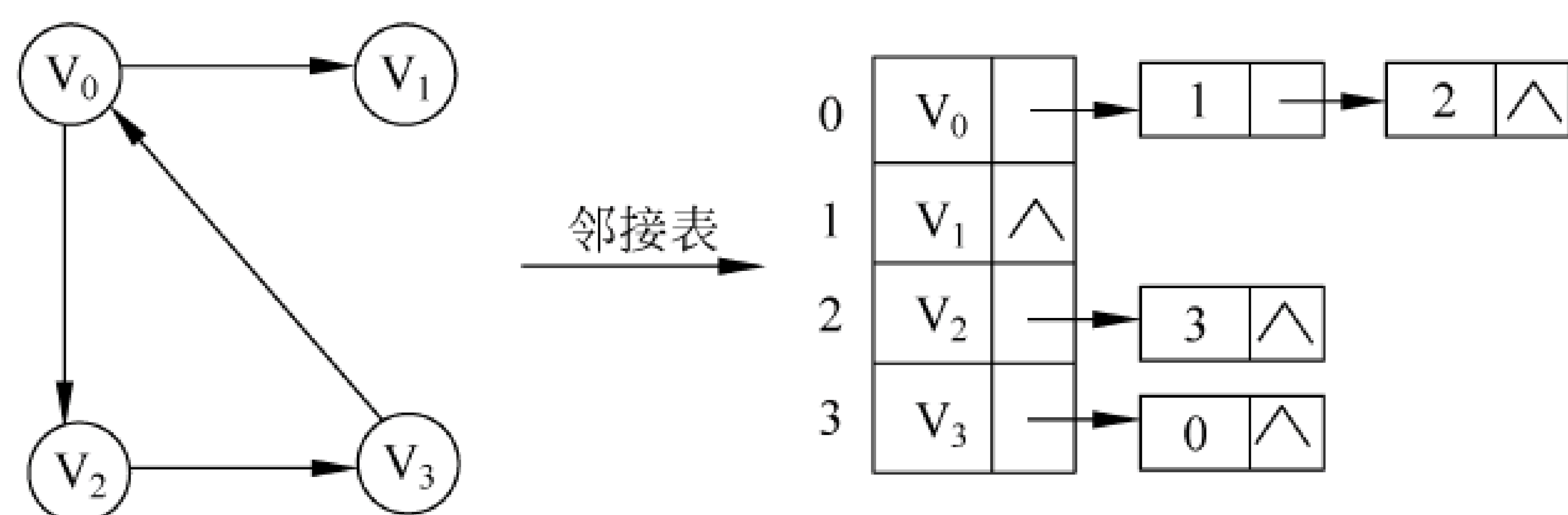
注意:邻接矩阵的特点如下。

- 图的邻接表表示不唯一。这是因为在每个顶点对应的单链表中,各边结点的链接次序可以是任意的,取决于建立邻接表的算法以及边的输入次序。
- 对于有 n 个顶点和 e 条边的无向图,其邻接表有 n 个表头结点和 2e 个边结点;对于有 n 个顶点和 e 条边的有向图,其邻接表有 n 个表头结点和 e 个边结点。显然,对于边的数目较少的稀疏图,邻接表比邻接矩阵节省空间。
- 无向图顶点 i(0≤i≤n−1)的度:等于顶点的 i 号链表的边结点数目。
- 有向图顶点 i(0≤i≤n−1)的出度:等于顶点的 i 号链表的边结点数目。
- 有向图顶点 i(0≤i≤n−1)的入度:等于邻接表中所有 adjvex 域值为 i 的边结点数。

图的邻接表存储类型的定义如下。



(a) 无向图的邻接表



(b) 有向图的邻接表

图 7.11　两个邻接表

```
typedef struct ArcNode
{ int adjvex;                        //顶点 i 相邻的顶点的编号
   struct ArcNode * nextarc;         //下一个边结点指针域
   InfoType info                     //该边的相关信息
}ArcNode;                            //边结点类型
typedef struct VNode
{ Vertex data;                       //顶点信息域
   ArcNode * firstarc;               //指向第一个边结点
}VNode;                              //邻接表头结点类型
typedef VNode AdjList[MAXV]; //AdjList 是邻接表类型
typedef struct ALGraph
{ AdjList adjlist;                   //邻接表
   int n,e;                          //顶点数 n 和边数 e
}ALGraph;                            //邻接表
```

由于在有向图的邻接表中只存放了以顶点为起点的边,所以不容易找到指向某一顶点的边。为此,可以设计有向图的逆邻接表。所谓**逆邻接表**,就是在有向图的邻接表中,对每个顶点,邻接指向进入该顶点的边。例如,图 7.11(b)的逆邻接表如图 7.12 所示。

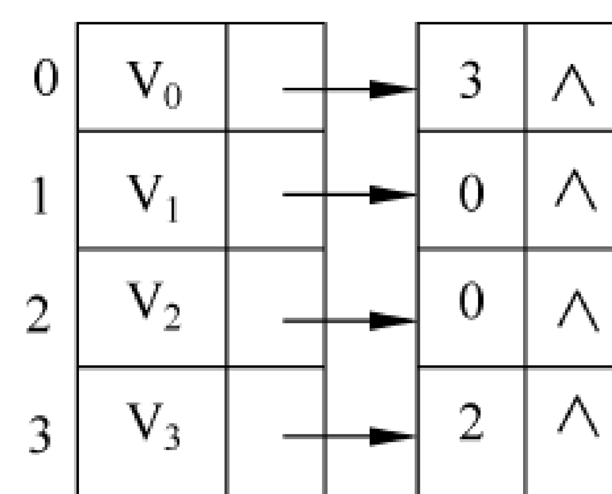


图 7.12　图 7.11(b)的逆邻接表

7.2.3　十字链表

十字链表是有向图的另一种链式存储结构。可以把十字链表看成是将有向图的邻接表和逆邻接表结合起来的一种有向图链式存储结构,即

- 每个顶点必有一个顶点结点。
- 有向图的每一条弧都有一个弧结点。

顶点结点：

data	firstin	firstout

弧结点：

tailvex	headvex	hlink	tlink	info

弧头相同的弧在同一链表上，弧尾相同的弧也在同一链表上，它们的头结点即顶点结点，由 3 个域组成：其中，data 域存储和顶点相关的信息，如顶点的名称等；firstin 和 firstout 为两个链域，分别指向以该顶点为弧头或弧尾的第一个弧结点。在弧结点中有 5 个域：其中，尾域(tailvex)和头域(headvex)分别指示弧尾和弧头这两个顶点在图中的位置；链域 hlink 指向弧头相同的下一条弧；链域 tlink 指向弧尾相同的下一条弧；info 域指向该弧的相关信息。例如，图 7.11(b)的十字链表如图 7.13 所示。

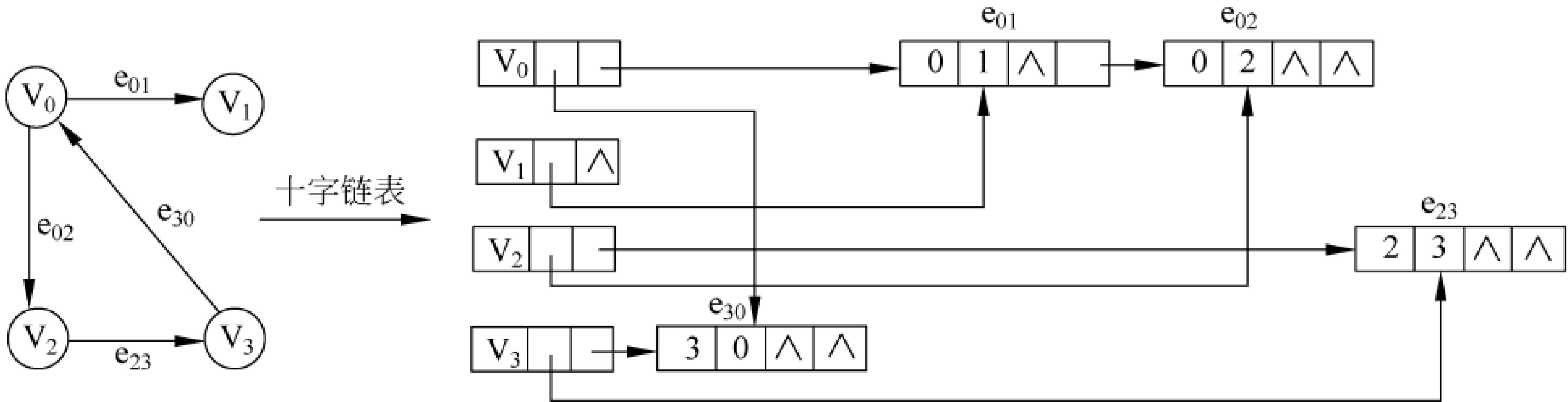


图 7.13　图 7.11(b)的十字链表

注意：十字链表的特点如下。

- 顶点结点数＝顶点数；弧结点数＝弧的条数。
- 求入度：从顶点 V_i 的 firstin 出发，沿着弧结点中的 hlink 所经过的弧结点数。
- 求出度：从顶点 V_i 的 firstout 出发，沿着弧结点中的 tlink 所经过的弧结点数。

有向图的十字链表存储类型的定义如下。

```
#define MAX_VERTEX_NUM 20
typedef struct ArcBox
{ int tailvex, headvex;                    //该弧的尾和头顶点的编号
  struct ArcBox  * hlink, * tlink;         //分别为弧头相同和弧尾相同的弧的链域
  InfoType  * info;                        //该弧相关信息的指针
}ArcBox;                                   //弧结点类型
typedef struct VexNode
{ VertexType data;
  ArcBox  * firstin, * firstout;           //分别指向该顶点第一条入弧和出弧
}VexNode;                                  //顶点结点类型
typedef struct
{ VexNode xlist[MAX_VERTEX_NUM];           //表头向量
  int n, e;                                //有向图的当前顶点数和弧数
}OLGraph;                                  //十字链表类型
```

【例 7.3】 对于具有 n 个顶点的不带权图 G，回答以下问题：

(1) 设计一个将邻接矩阵转换为邻接表的算法。

(2) 设计一个将邻接表转换为邻接矩阵的算法。

(3) 分析上述两个算法的时间复杂度。

解：(1) 在邻接矩阵上查找值不为 0 的元素，找到这样的元素后创建一个表结点，并在邻接表对应的单链表中采用首插法插入该结点。算法如下。

```
void MatToList(MGraph g, ALGraph * &G)           //将邻接矩阵 g 转换成邻接表 G
{ int i,j;
 ArcNode * p;
 G=(ALGraph * )malloc(sizeof(ALGraph));
 for(i=0;i<g.n;i++)                              //给邻接表中所有头结点的指针域置初值
    G->adjlist[i].firstarc=NULL;
 for(i=0;i<g.n;i++)                              //检查邻接矩阵中的每个元素
    for(i=g.n-1;i>=0;i--)
      if(g.edges[i][j]!=0)                       //存在一条边
      { p=(ArcNode * )malloc(sizeof(ArcNode));   //创建一个结点 * p
        p->adjvex=j;
        p->nextarc=G->adjlist[i].firstarc;       //采用首插法插入 * p
        G->adjlist[i].firstarc=p;
        }
    G->n=g.n;G->e=g.e;
}
```

(2) 假设邻接矩阵 g 中所有的元素值均为 0，然后在邻接表中查找顶点 i 的相邻结点 * p，找到后将邻接矩阵 g. edges[i][p->adjvex]的值修改为 1。算法如下。

```
void ListToMat(ALGraph * G, MGraph &g)            //将邻接表 G 转换成邻接矩阵 g
{ int i;
 ArcNode * p;
 for(i=0;i<G->n;i++)
 { p=G->adjlist[i].firstarc;
  while(p!=NULl)
  { g.edges[i][p->adjvex]=1;
    p=p->nextarc;
   }
 }
g.n=G->n;g.e=G->e;
}
```

(3) 算法(1)中有两重 for 循环，其时间复杂度为 $O(n^2)$。算法(2)中虽有两重 for 循环，但只对邻接表的表头结点和边结点访问一次，对于无向图，其时间复杂度为 O(n+2e)，对于无向图，其时间复杂度为 O(n+e)，其中 e 为图的边数。

注意：本例适合不带权的有向图和无向图，对于带权图邻接矩阵和邻接表的相互转换，只针对带权图特点稍做修改即可。

7.3 图的遍历与连通性

与之前章节讲过的树的遍历类似,我们希望从给定图中任意指定的顶点(称为初始点)出发,按照某种搜索方法沿着图的边访问图中的所有顶点,使每个顶点仅被访问一次,这个过程称为**图的遍历**。图的遍历算法是求解图的连通性问题、拓扑排序和求关键路径等算法的基础。

图的遍历比树的遍历更复杂,因为从树根到达树中的任意结点只有一条路径,而从图的初始点到达图中的某个顶点可能存在多条路径。当沿着图中的一条路径访问过某一顶点后,可能还会沿着另一条路径回到该顶点,即存在**回路**。为了避免同一个顶点被重复访问,必须记住每个被访问过的顶点。为此,可设置一个访问标志数组 visited[ ],当顶点 i 被访问过时,数组中的元素 visited[i]为 1;否则为 0。

根据搜索方法的不同,图的遍历方法有两种:一种叫**深度优先遍历**(Depth First Traverse)方法;另一种叫**广度优先遍历**(Breadth First Traverse)方法。


视频讲解

7.3.1 深度优先遍历

深度优先遍历(DFS)类似于树的先根遍历,是树的先根遍历的推广。

深度优先遍历的基本思想如下。

step1:从图中某个顶点 v 出发,访问此顶点,并将其作为当前顶点 v。

step2:搜索当前顶点 v 的下一个未被访问的邻接点;重复执行 step1 和 step2,直至当前顶点 v 的邻接点均被访问。

step3:若此时图中尚有顶点未被访问,则沿搜索路径回退,退到有邻接点未被访问的顶点,将该顶点作为当前顶点,重复上述步骤,直到所有顶点均被访问为止。

例如,对于图 7.8 所示的有向图,从顶点 0 开始进行深度优先遍历,可以得到如下访问序列:0 1 2 4 3 或 0 3 2 4 1。

以邻接表为存储结构的深度优先遍历算法如下(其中,v 是初始顶点编号,visited[ ]是一个全局数组,初始时所有元素均为 0,表示所有顶点均未被访问过)。

```
void DFS(ALGraph *G,int v)
{ ArcNode *p;                          //以 v 为起始点用邻接表进行 DFS 搜索
  visited[v]=1;                        //访问顶点 v,并作标记
  printf("%d",v);                      //输出被访问顶点的编号
  p=G->adjlist[v].firstarc;            //p 指向顶点 v 的第一个邻接点
  while(p!=null)                       //依次搜索 v 的邻接点
  { if (visited[p->adjvex]==0)         //若邻接点未被访问过,则递归访问它
          DFS(G,p->adjvex);            //递归返回
  p=p->nextarc;                        //找 Vi 的下一个邻接点
 }
}
```

以邻接矩阵为存储结构的深度优先遍历算法与此类似,这里不再列出。

例如,以图 7.11(a)的邻接表为例调用 DFS()函数,假设初始顶点编号 $v=V_2$,调用 DFS(G,V_2)的执行过程如下。

（1）DFS(G,V_2)：访问顶点 V_2，找顶点 V_2 的相邻顶点 V_4，它未被访问过，转(2)。

（2）DFS(G,V_4)：访问顶点 V_4，找顶点 V_4 的相邻顶点 V_2，它已被访问，找下一个相邻顶点 V_1，它未被访问过，转(3)。

（3）DFS(G,V_1)：访问顶点 V_1，找顶点 V_1 的相邻顶点 V_4、V_2，它们均已被访问，找下一个相邻顶点 V_0，它未被访问过，转(4)。

（4）DFS(G,V_0)：访问顶点 V_0，找顶点 V_0 的相邻顶点 V_3，它未被访问过，转(5)。

（5）DFS(G,V_3)：访问顶点 V_3，找顶点 V_3 的相邻顶点，所有相邻顶点均已被访问，退出 DFS(G,V_3)，转(6)。

（6）继续 DFS(G,V_0)：顶点 V_0 的所有后继相邻顶点均已被访问，退出 DFS(G,V_0)，转(7)。

（7）继续 DFS(G,V_1)：顶点 V_1 的所有后继相邻顶点均已被访问，退出 DFS(G,V_1)，转(8)。

（8）继续 DFS(G,V_4)：顶点 V_4 的所有后继相邻顶点均已被访问，退出 DFS(G,V_4)，转(9)。

（9）继续 DFS(G,V_2)：顶点 V_2 的所有后继相邻顶点均已被访问，退出 DFS(G,V_2)，转(10)。

（10）结束。

如图 7.14 所示，从顶点 V_2 出发的深度优先访问序列是：$V_2V_4V_1V_0V_3$。

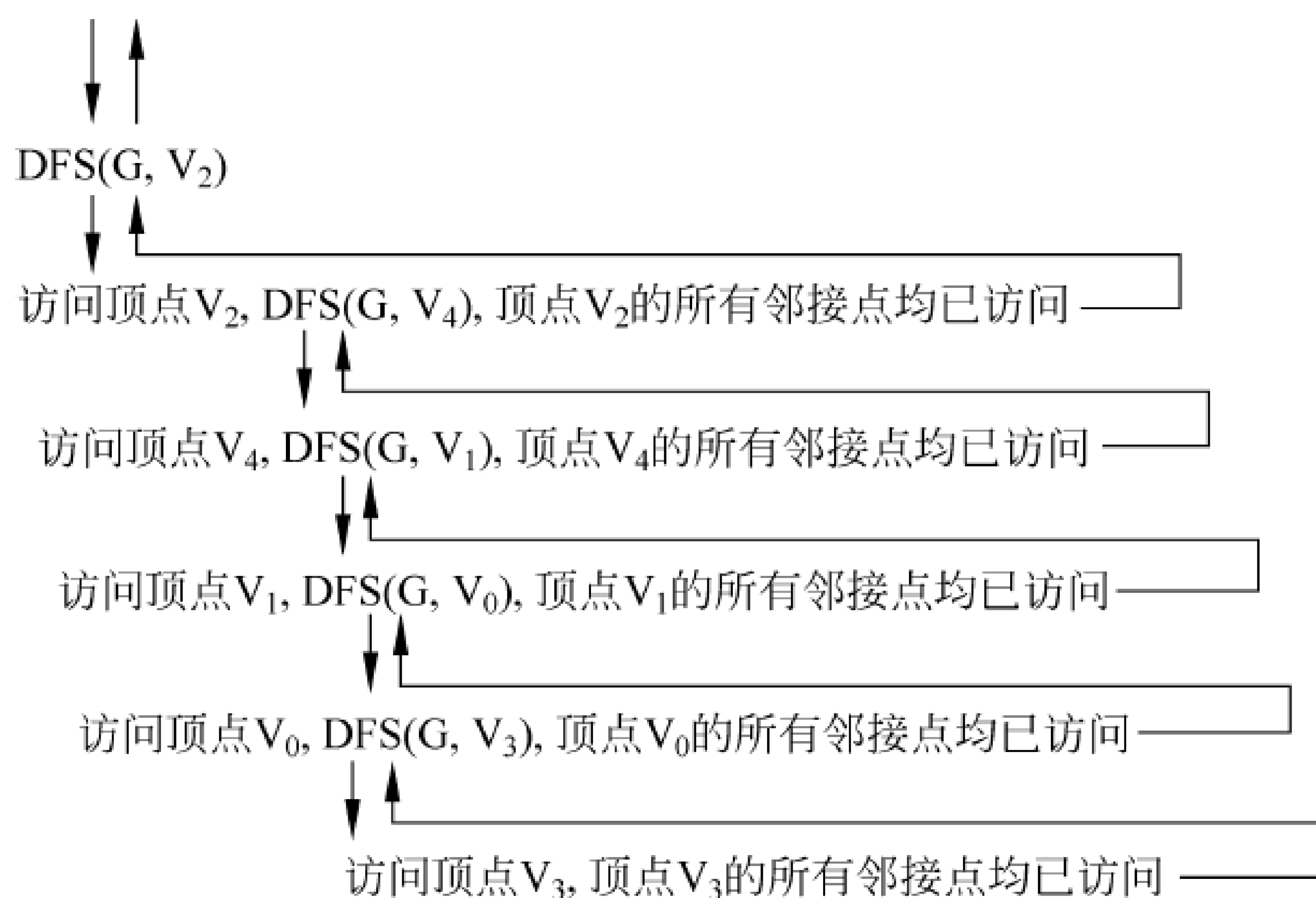


图 7.14　DFS(G,V_2)的执行过程

对于有 n 个顶点 e 条边的有向图或无向图，DFS 算法对图中每个顶点至多调用一次，因此其递归调用总次数为 n。

当访问某个顶点 v 时，DFS 的时间主要花在从该顶点出发搜索它的邻接点上。用邻接表表示图时，需遍历该顶点对应单链表中的所有邻接点，所以 DFS 的总时间为 $O(n+e)$；当用邻接矩阵表示图时，需遍历该顶点对应行的所有 n 个元素，所以 DFS 的总时间为 $O(n^2)$。

7.3.2　广度优先遍历



视频讲解

广度优先遍历(BFS)类似于树的层次遍历。广度优先遍历的基本思路如下。

step1：从图中某顶点 v 出发，将 v 作为当前顶点。

step2：依次访问当前顶点 v 的所有未被访问的邻接点，然后分别从这些邻接点出发依次访问它们的未被访问的邻接点，使得先被访问的顶点的邻接点先被访问。

step3：重复 **step2**，直到图中的所有顶点都被访问。

例如,对于图 7.8 所示的有向图,从顶点 0 开始进行广度优先遍历,可以得到如下访问序列:0 1 3 2 4 或 0 3 1 2 4。

以邻接表为存储结构,用广度优先遍历图时,需要使用一个队列,以类似于层次遍历二叉树的方式遍历图。以邻接矩阵为存储结构的广度优先遍历算法与此类似,这里不再列出。

例如,以图 7.11(a)的邻接表为例调用 BFS(G,v)函数,假设初始顶点编号 $v=V_2$,调用 BFS(G,V_2)的执行过程如下。

对应的算法如下(其中,v 是初始顶点编号)。

```
void BFS(ALGraph *G,int v)
{ ArcNode *p;
 int queue[MAVX],front=0,rear=0;           //定义循环队列并初始化队头和队尾
 int visited[MAVX];                        //定义存放顶点的访问标志的数组
 int w,i;
 for (i=0; i<G->n;i++)
       visited[i]=0;                       //访问标志数组初始化
 printf("%2d",v);                          //输出 i 访问顶点的编号
 visited[v]=1;                             //置已访问标志
 rear=(rear+1)%MAVX;
 queue[rear]=v;
 while(front!=rear)                        //v 进队
 { front=(front+1)%MAVX;                   //若队列不空时,循环
   w=queue[front];                         //出队并赋给 w
   p=G->adjlist[w].firstarc;               //找顶点 w 的第一个邻接点
   while(p!=NULL)
   { if(visited[p->adjvex]==0)             //若当前邻接顶点未被访问
      { printf("%2d",p->adjvex);           //访问相邻顶点
      visited[p->adjvex]=1;                //置该顶点已被访问的标志
      rear=(rear+1)%MAVX;                  //该顶点进队
      queue[rear]=p->adjvex;
      }
     p=p->nextarc;                         //找顶点 w 的下一个邻接点
   }
 }
 printf("\n");
}
```

(1) 访问顶点 V_2,V_2 入队,转(2)。

(2) 第 1 次循环:顶点 V_2 出队,找它的第一个相邻顶点 V_1,它未被访问过,访问它并将 V_1 入队;找 V_2 的下一个相邻顶点 V_3,它未被访问过,访问它并将 V_3 入队;找 V_2 的下一个相邻顶点 V_4,它未被访问过,访问它并将 V_4 入队,转(3)。

(3) 第 2 次循环:顶点 V_1 出队,找它的第一个相邻顶点 V_0,它未被访问过,访问它并将 V_0 入队;找 V_1 的下一个相邻顶点 V_4,它被访问过,转(4)。

(4) 第 3 次循环:顶点 V_3 出队,找它的第一个相邻顶点 V_2,它被访问过;找 V_3 的下一个相邻顶点 V_0,它未被访问过,访问它并将 V_0 入队,转(5)。

(5) 第 4 次循环:顶点 V_4 出队,依次找其相邻顶点 V_1、V_2,它们均被访问过,转(6)。

(6) 第 5 次循环:顶点 V_0 出队,依次找其相邻顶点 V_1、V_3,它们均被访问过,转(7)。

(7) 此时队列为空,遍历结束。

如图 7.15 所示，从顶点 V_2 出发的广度优先访问序列是：$V_2V_1V_3V_4V_0$。

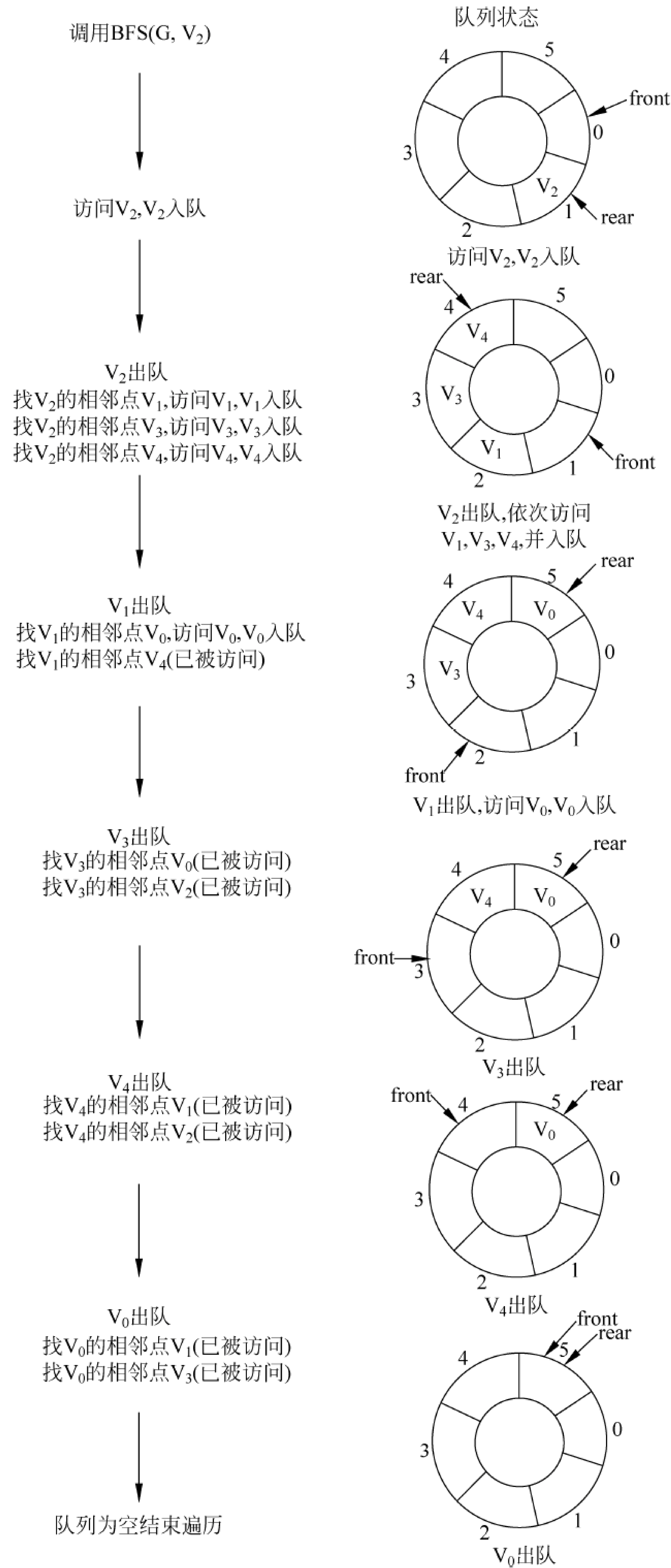


图 7.15　BFS(G,V_2)的执行过程

对于具有 n 个顶点 e 条边的有向图或无向图，BFS 算法中每个顶点入队一次，因此执行时间与 DFS 相同。当图采用邻接表表示时，BFS 的总时间为 O(n+e)；当图采用邻接矩阵表示时，BFS 的总时间为 $O(n^2)$。

7.3.3 连通分量

1. 无向图的连通分量

上面讨论的图的两种遍历方法在对无向图进行遍历时，若无向图是连通图，则一次遍历能够访问到图中的所有顶点；但若无向图是非连通图，则需从多个顶点出发进行搜索，而每一次从一个新的起始点出发进行搜索过程中得到的顶点访问序列恰为其各个连通分量中的顶点集。

2. 有向图的强连通分量

对有向图进行遍历时，若从初始点到图中的每个顶点都有路径，则一次遍历能够访问到图中的所有顶点；否则一次遍历不能访问到图中的所有顶点，为此同样需要再选初始点，继续遍历，直到图中所有顶点都被访问过为止。

采用深度优先遍历非连通无向图的算法如下。

```
DFS1(ALGraph *G)
{ int i;
 for(i=0;i<G->n;i++)
   if(visited[i]==0)
       DFS(G,i);
}
```

采用广度优先遍历非连通无向图的算法如下。

```
BFS1(ALGraph *G)
{ int i;
 for(i=0;i<G->n;i++)
   if(visited[i]==0)
       BFS(G,i);
}
```

【例 7.4】 基于深度优先遍历算法的应用。

假设图 G 采用邻接表存储，设计一个算法，判断图 G 中从顶点 u 到顶点 v 是否存在简单路径。

解：简单路径是指路径上的顶点不重复。采用深度优先遍历的方法，从顶点 u 到顶点 v 的深度优先遍历过程如图 7.16 所示。为此，在深度优先遍历算法的基础上增加 v 和 has 两个形参，其中 has 表示顶点 u 到顶点 v 是否有路径，其初值为 false，当从顶点 u 遍历到顶点 v 后，置 has 为 true 并返回。查找从顶点 u 到顶点 v 是否存在简单路径的过程如图 7.17 所示。

对应的算法如下。

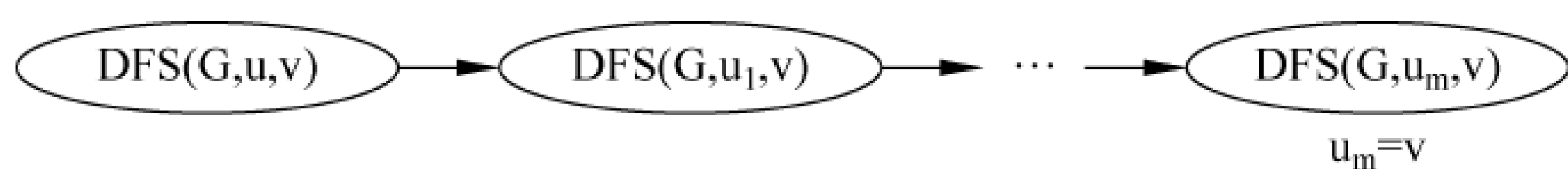


图 7.16　从顶点 u 到顶点 v 的深度优先遍历过程

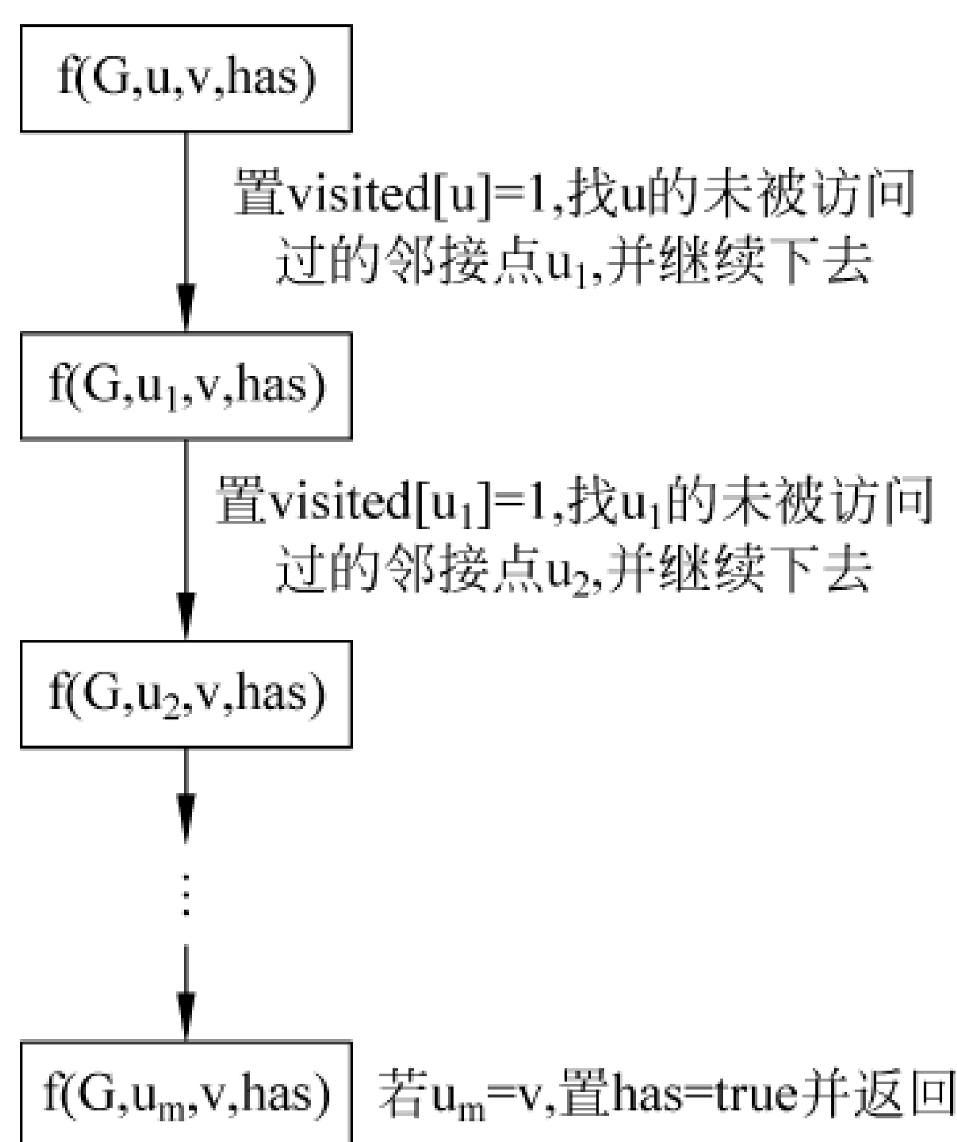


图 7.17　查找从顶点 u 到顶点 v 是否存在简单路径的过程

```
void ExistPath(AGraph *G,int u,int v,bool &has)
{ //has 表示顶点 u 到顶点 v 是否有路径,初值为 false
  int w;
  ArcNode *p;
  visited[u]=1;                  //置已访问标志
  if(u==v)                       //找到了一条路径
  { has=true;                    //置 has 为 true 并结束算法
    return;
  }
  p=G->adjlist[u].firstarc; //p 指向顶点 u 的第一个相邻点
  while(p!=NULL)
  { w=p->adjvex;                 //w 为顶点 u 的相邻顶点
if(visited[w]==0)                //若 w 顶点未被访问,则递归访问它
        ExistPath(G,w,v,has);
p=p->nextarc;                    //p 指向顶点 u 的下一个相邻点
  }
}
```

【例 7.5】 基于广度优先遍历算法的应用。

假设图 G 采用邻接表存储,设计一个算法,求不带权无向连通图 G 中从顶点 u 到顶点 v 的一条最短路径。

解:图 G 不带权的无向连通图,一条边的长度计为 1,因此,求从顶点 u 到顶点 v 的最短路径即求从顶点 u 到顶点 v 的经过边数最少的顶点序列。利用广度优先遍历算法,从顶点 u 出发进行广度遍历,类似于从顶点 u 出发一层一层地向外扩展,当第一次找到顶点 v 时,队列中便包含了从顶点 u 到顶点 v 最短的路径,如图 7.18 所示,再利用队列输出最短路

径(逆路径)。由于要利用队列找出路径,所以采用非环形队列。

对应的算法如下。

```
typedef struct
{ int data;                          //顶点编号
  int parent;                        //前一个顶点的位置
}QUERE;                              //非环形队列的类型
void ShortPath(ALGraph *G,int u,int v)
{ //输出从顶点 u 到顶点 v 的最短逆路径
  ArcNode *p;
  int w,i;
  QUERE qu[MAVX];                    //非环形队列
  int front=-1,rear=-1;              //队列的头、尾指针
  int visited[MAVX];
  for (i=0;i<G->n;i++)               //访问标记置初值 0
      visited[i]=0;
  rear++;                            //顶点 u 进队
  qu[rear].data=u;
  qu[rear].parent=-1;
  visited[u]=1;
  while (front!=rear)                //队不空时循环
  { front++;                         //出队顶点 w
    w=qu[front].data;
    if(w==v)                         //找到 v 时输出路径之逆并退出
    { i=front;                       //通过队列输出逆路径
      while(qu[i].parent!=-1)
      { printf("%2d",qu[i].data);
        i=qu[i].parent;
      }
      printf("%2d\n",qu[i].data);
      break;
      }
   p=G->adjlist[w].firstarc;         //找 w 的第一个邻接点
   while(p!=NULL)
   { if(visited[p->adjvex]==0)
     { visited[p->adjvex]=1;
       rear++;                       //将 w 的未被访问过的邻接点入队
       qu[rear].data=p->adjvex;
       qu[rear].parent=front;
     }
     p=p->nextarc;                   //找 w 的下一个邻接点
    }
  }
}
```

【例 7.6】 有一个连通图 G,如图 7.19 所示,从顶点 a 出发对 G 进行深度优先遍历和广度优先遍历,请写出一种遍历结果,并画出由遍历过程得到的深度优先遍历生成树和广度优先遍历生成树。

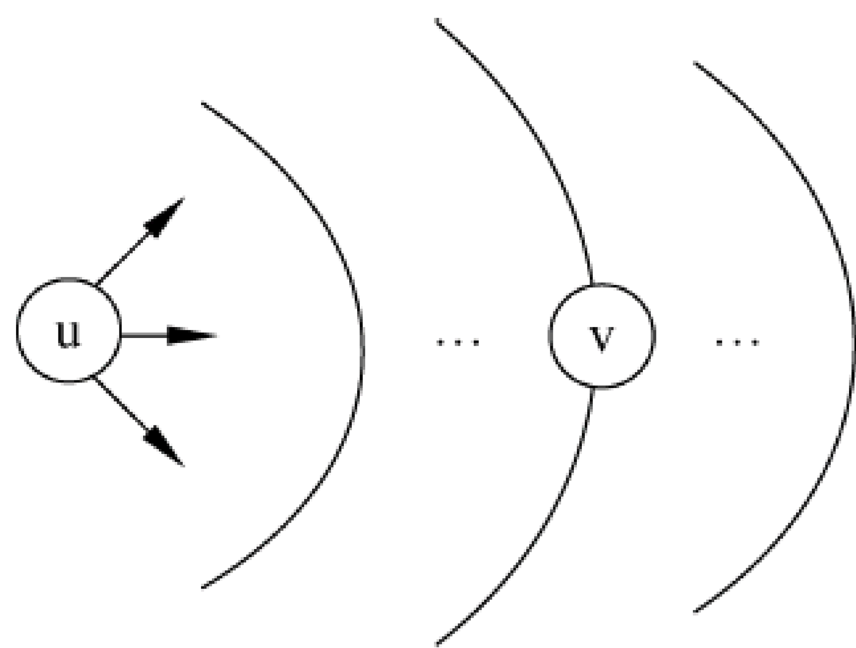


图 7.18　查找顶点 u 到顶点 v 的最短路径

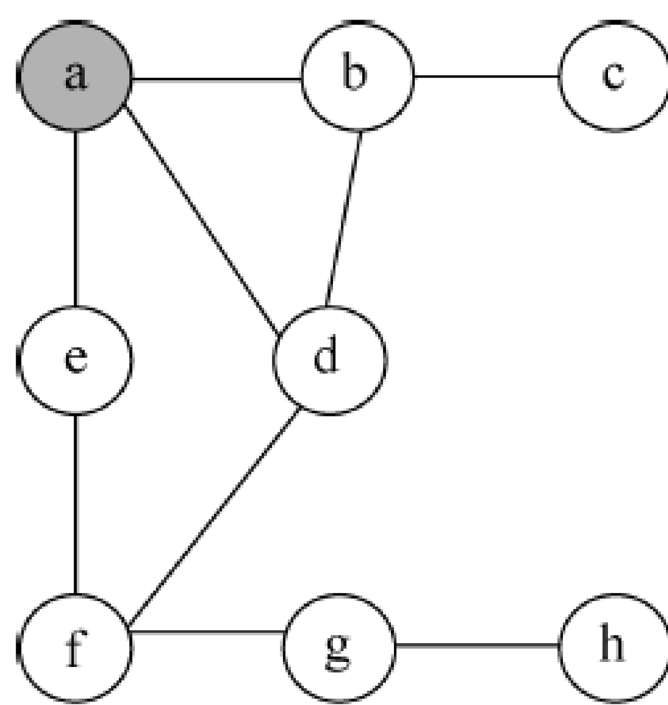


图 7.19　连通图 G

分析：从 a 顶点出发访问 a 的一个邻接点(b,d,e 均可以),若选择 e 顶点,则再从 e 顶点出发访问 e 的一个未被访问的邻接点 f,接着从 f 顶点出发依次类推访问 g 顶点,由 g 顶点访问 h 顶点,此刻 h 顶点已经没有未被访问的邻接点了,但图的遍历尚未结束；于是回退,由 h 顶点回退到 g 顶点,检查 g 顶点是否有未被访问的邻接点,若有,则访问,否则接着回退,由 g 顶点退回到 f 顶点,f 顶点有未被访问的邻接点 d,则访问 d 顶点,依次类推,访问 b 顶点,由 b 顶点访问 c 顶点；深度优先遍历结束。于是,深度优先遍历的一种结果是：a,e,f,g,h,d,b,c。

对于广度优先遍历,从 a 顶点出发访问 a 的所有邻接点 b,d,e(邻接点次序可任意),依次(按照 b,d,e 的次序)访问 b 顶点的所有未被访问的邻接点 c,d 顶点的所有未被访问的邻接点 f,e 没有未被访问的邻接点,依次类推先访问 c 的所有未被访问的邻接点(不存在),再访问 f 的所有未被访问的邻接点 g,g 顶点的所有未被访问的邻接点 h；广度优先遍历结束。于是,广度优先遍历的一种结果是：a,b,d,e,c,f,g,h。两种遍历结果对应的生成树分别如图 7.20 和图 7.21 所示。

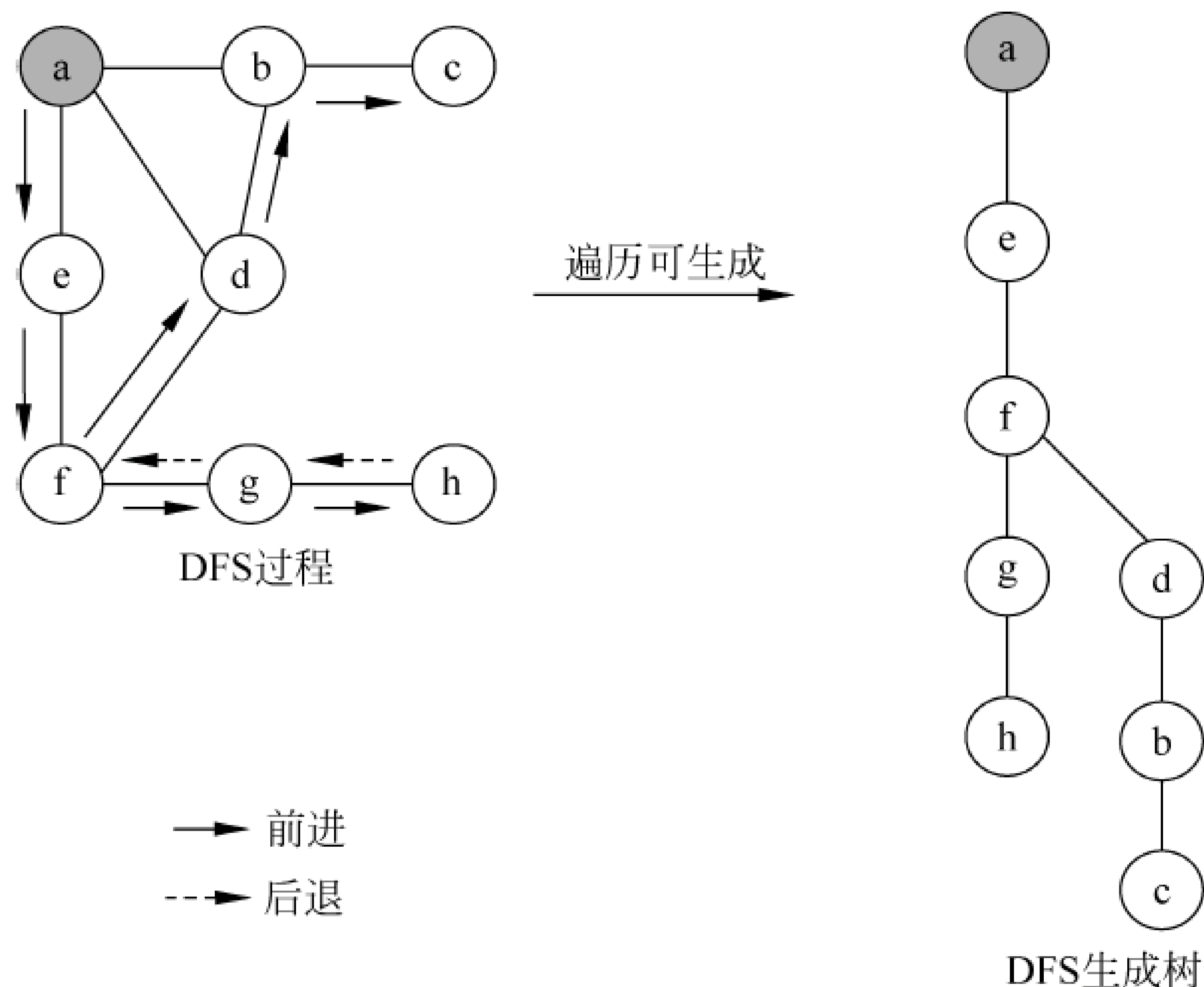


图 7.20　DFS 及 DFS 生成树

注意：图的深度优先遍历类似于树的先根遍历,遍历过程中有回溯现象,遍历过程一般不唯一,因此生成树的形态也不唯一。

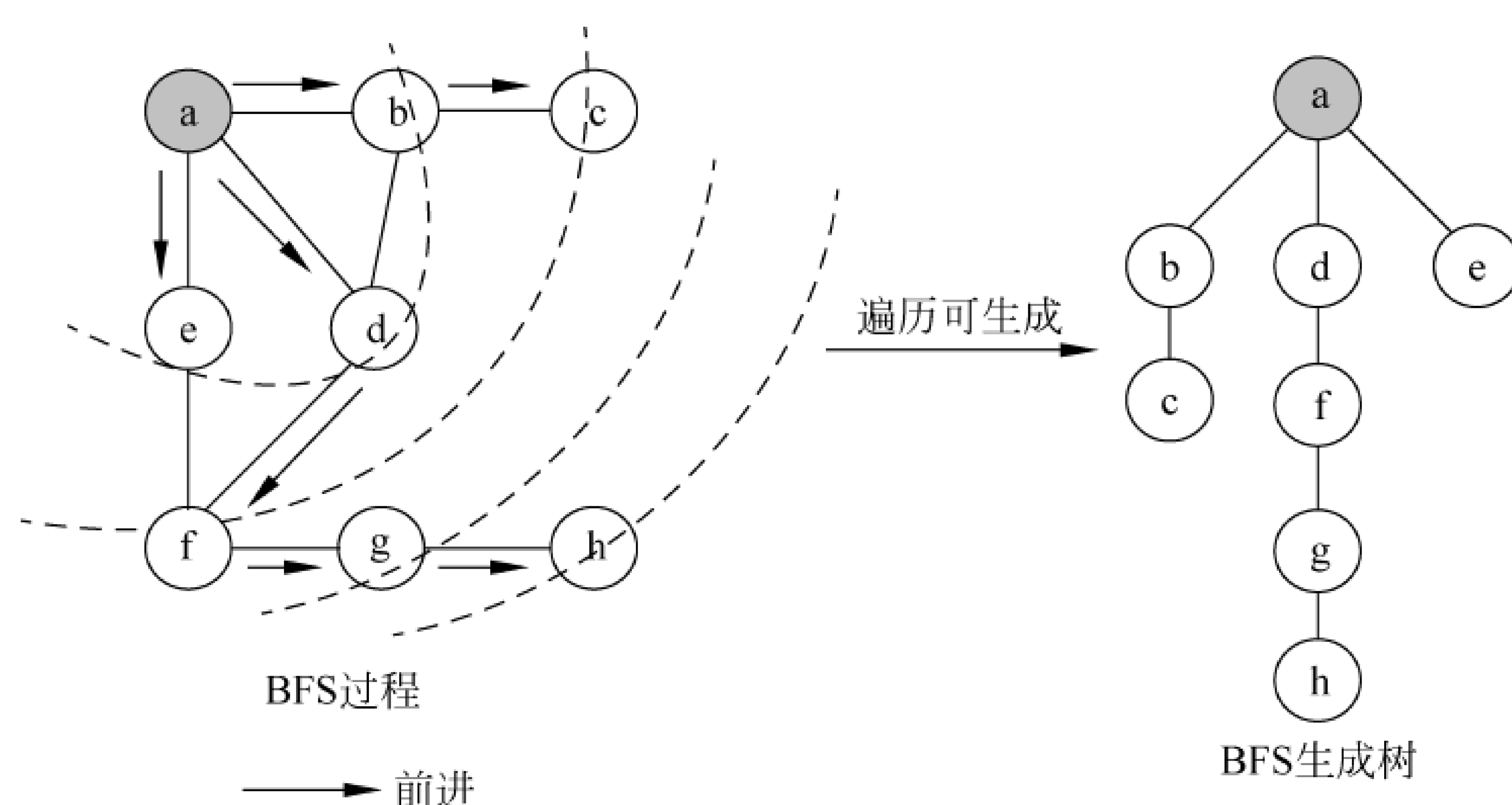


图 7.21　BFS 及 BFS 生成树

注意：图的广度优先遍历类似于树的层次遍历，遍历过程中没有回溯现象，遍历过程一般不唯一，因此生成树的形态也不唯一。一般而言，DFS 生成树的高度大于等于 BFS 生成树的高度。

7.4　最小生成树

生成树：一个连通图的生成树是该连通图的一个极小连通子图，它含有图中的全部顶点，但只有构成一棵树的(n－1)条边。

如果在一棵生成树上添加一条边，必定构成一个环，因为这条边使得它依附的另两个顶点之间有了第二条路径。一个图有 n 个顶点，如果它有小于(n－1)条边，则是非连通图；如果它有多于(n－1)条边，则一定有回路。

注意：一棵有 n 个顶点的生成树(连通无回路图)有且仅有(n－1)条边，但是，有(n－1)条边的图不一定都是生成树。

对于一个带权(假定每条边上的权值均为大于 0 的实数)连通无向图 G 中的不同生成树，各棵树的边上的权值之和可能不同，边上的权值之和最小的树称为该图的**最小生成树**。

按照生成树的定义，n 个顶点的连通图的生成树有 n 个顶点，(n－1)条边。因此，构造最小生成树的准则有以下 3 条。

- 尽可能使用该图中权值小的边构造最小生成树。
- 必须使用且仅使用(n－1)条边连接图中的 n 个顶点。
- 不能使用产生回路的边。

7.4.1　普里姆算法



视频讲解

普里姆(Prim)算法(选点法)是一种构造性算法。

基本思路如下。

step1：取图中顶点 v 作生成树的根，之后按 **step2** 的要求向生成树 T 添加顶点。

step2：依次选取一端不在树 T(即集 V－T)中，另一端已在树 G 中，且权值最小的边，把该边和顶点分别并入树 T 的边集 TE 和顶点集中。

step3：重复执行 **step2**，直到把 n 个顶点都并入树 T 的顶点集中(共选取 n−1 条边)。

假设 G=(V,E)为一带权图，其中 V 为带权图中所有顶点的集合，E 为带权图中所有带权边的集合。设置两个新的集合 U 和 TE，其中集合 U 用于存放 G 的最小生成树中的顶点，集合 TE 用于存放 G 的最小生成树中的边。令集合 U 的初值为 U={u}(假设构造最小生成树时，从顶点 u 出发)，集合 TE 的初值为 TE={}。按照 Prim 算法的思想，从所有 u∈U，v∈V−U 的边中选取具有最小权值的边(u,v)，将顶点 v 加入集合 U 中，将边(u,v)加入集合 TE 中，如此不断重复，直到 U=V 时，最小生成树构造完毕，这时集合 TE 中包含了最小生成树的所有边。

Prim 算法可用下述过程描述，其中用 w_{uv} 表示顶点 u 与顶点 v 边上的权值。

```
step1:U={u},TE={};
step2:while (U≠V)do
(u,v)=min{w_uv;u∈U,v∈V−U}
TE=TE+{(u,v)}
U=U+{v}
step3:结束.
```

注意：生成树的边集 TE 初始值是一个空集。

计算机科学家简介：

Robert Clay Prim(1921—)，美国数学家和计算机科学家，1941 年获得电气工程学士，1949 年从普林斯顿大学获得数学博士学位。1941～1944 年，他担任通用电气工程师。1944～1949 年，他于美国海军军械实验室担任工程师，后来成为数学家。1958～1961 年，他在贝尔实验室时担任数学研究部主任，1957 年提出了 Prim 算法。

下面的 Prim(g,v)算法依照上述过程构造最小生成树，其中的参数 g 为带权邻接矩阵，v 为开始顶点的编号。对于带权无向图，邻接矩阵 g. edges 的定义为

$$g.\ edges[i][j] = \begin{cases} w_{ij} & \text{当 } i \neq j \text{ 且}(i,j) \in E(G) \\ 0 & \text{当 } i = j \\ \infty & \text{其他} \end{cases}$$

假设图中各边权值大于 0，远小于 32767，因此∞值采用 32767。

为了便于在集合 U 和 V−U 之间选择权值最小的边，可建立两个数组 closest 和 lowcost，它们记录从 U 到 V−U 权值最小的边。对于某个 j∈V−U，closest[j]存储该边依附在 U 中的顶点编号，lowcost[j]存储该边的权值，如图 7.22 所示，其意义为：若 lowcost[j]=0，则表明顶点 j∈U；若 0<lowcost[j]<∞，则表明顶点 j∈V−U，且顶点 j 和 U 中的顶点 closest[j]构成的边(j,closest[j])是所有与顶点 j 相邻、另一端在 U 的边中权值最小的边，其最小权值为 lowcost[j](对于每个顶点 j∈V−U，U 中的顶点到顶点 j 可能有多条边，但只有一个最小边，用 closest[j]表示对应顶点，用 lowcost[j]表示该边的权值；若 lowcost[j]=∞，则表示顶点 j 与 closest[j]之间没有边。

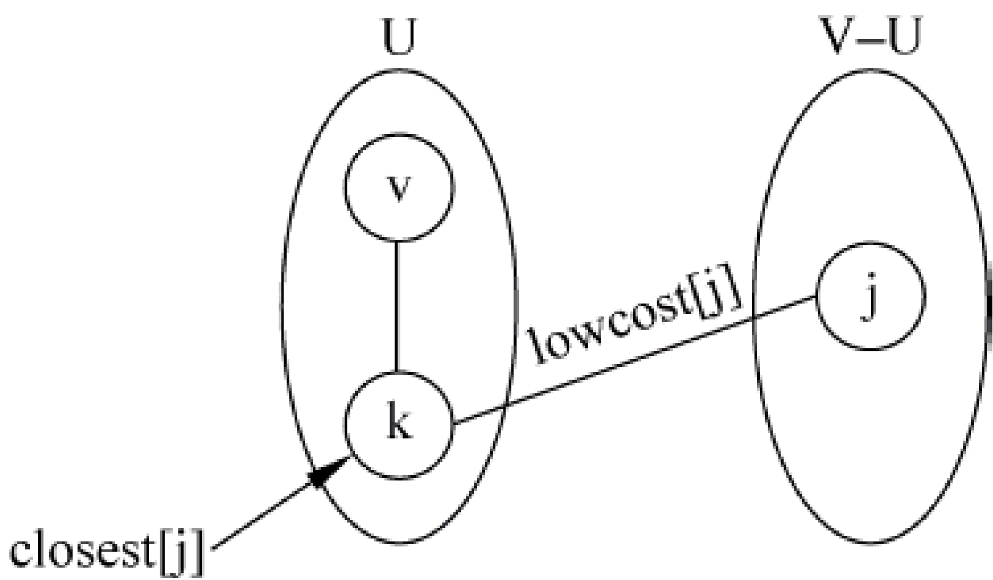


图 7.22　顶点集合 U 和 V−U

```
#define INF 32767
void Prim(MGraph g,int v)
{ int lowcost[MAVX];
    int min;
    int closest[MAVX],i,j,k;
    for (i=0;i<g.n;i++)                    //给 lowcost[ ]和 closest[ ]置初值
    { lowcost[i]=g.edges[v][i];
        closest[i]=v;
     }
    for (i=1;i<g.n;i++)                    //找出(n-1)个顶点
    { min=INF;
     for (j=0;j<g.n;j++)                   //在(V-U)中找出离 U 最近的顶点 k
        if(lowcost[j]!=0&&lowcost[j]<min)
        { min=lowcost[j];
         K=j;                              //k 记录最近顶点的编号
        }
    printf("边(%d,%d)权为:%d\n",closest[k],k,min);
    lowcost[k]=0;                          //标记 k 已经加入 U
   for (j=0;j<g.n;j++)                     //修改数组 lowcost 和 closest
     if(g.edges[k][j]!=0&&g.edges[k][j]<lowcost[j])
     { lowcost[j]=g.edges[k][j];
        closest[j]=k;
        }
    }
}
```

图 7.23 为带权无向图调用 Prim 算法求解最小生成树的过程，其中邻接矩阵如下。

{{0,6,1,5,INF,INF},{6,0,5,INF,3,INF},{1,5,0,5,6,4},{5,INF,5,0,INF,2},{INF,3,6,INF,0,6},{INF,INF,4,2,6,0}}

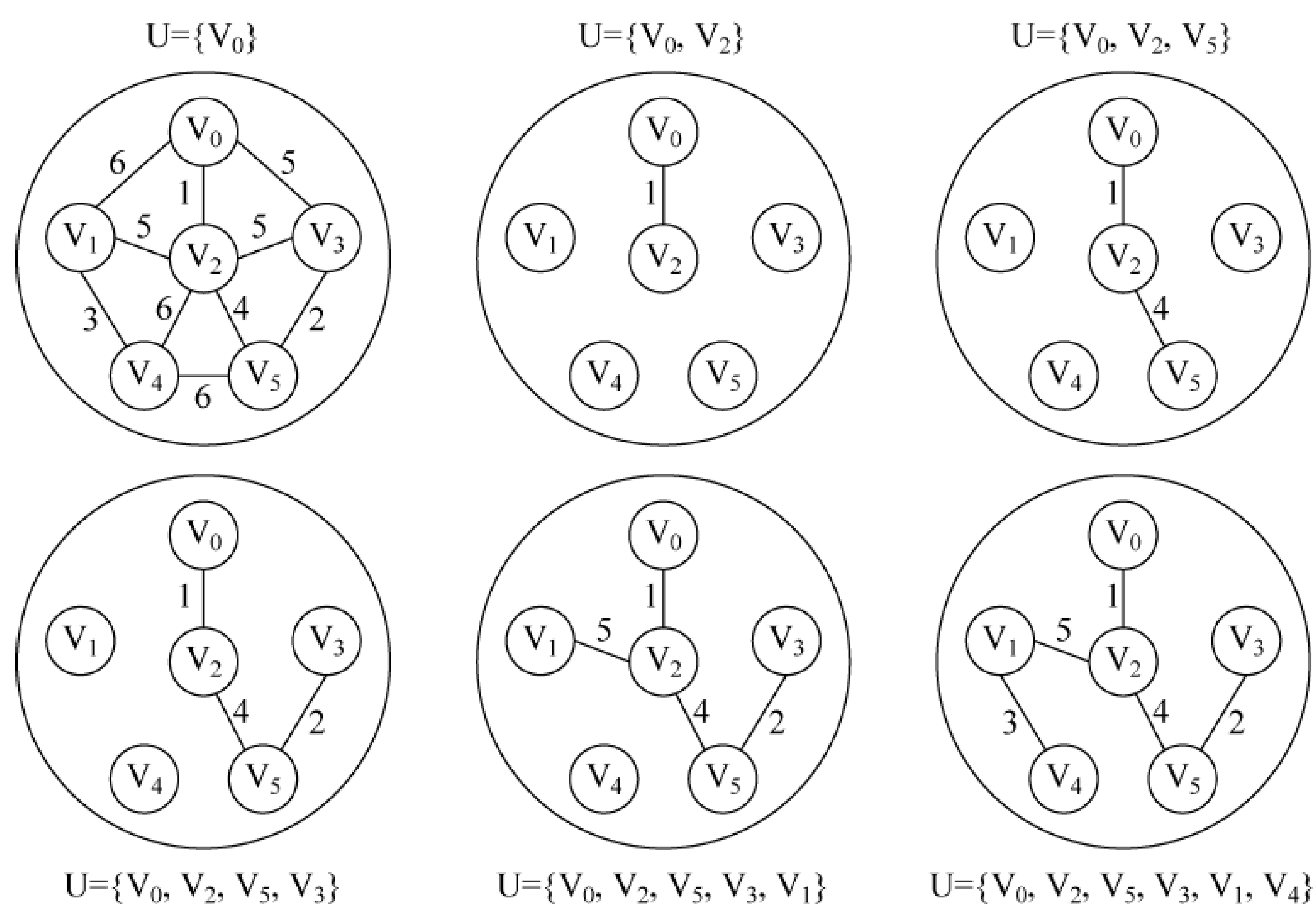


图 7.23 带权无向图调用 Prim 算法求解最小生成树的过程

在算法中，lowcost 数组记录从 U 中顶点到 V－U 中顶点候选边的权值，其目的是为了求出最小边，该数组只需有 n 个元素空间。首先保存顶点 v 到其他(n－1)个顶点的边的权值，共 n 个元素(含 v 到 v 的边，其权值为 0)，从中选取一条边(v，k)(从 lowcost 数组中选取权值不为 0 的最小者)，并将 k 对应的元素 lowcost[k]置为 0，表示将顶点 k 加入到 U 中。然后从顶点 k 出发进行类似的操作，直到选取 n－1 条边。例如，对于图 7.23，Prim 算法求解过程中 lowcost 数组的变化见表 7.1。

表 7.1　Prim 算法求解过程中 lowcost 数组的变化

起点	lowcost 数组保存的候选边及其权值	选择的边(不为 0 的权值中最小者)
V_0	(V_0,V_0)：0 (V_0,V_1)：6 (V_0,V_2)：1 (V_0,V_3)：5 (V_0,V_4)：∞ (V_0,V_5)：∞	(V_0,V_2)：1
V_2	(V_2,V_0)：0 (V_2,V_1)：5 (V_2,V_2)：0 (V_2,V_3)：5 (V_2,V_4)：6 (V_2,V_5)：4	(V_2,V_5)：4
V_5	(V_5,V_0)：0 (V_5,V_1)：5 (V_5,V_2)：0 (V_5,V_3)：2 (V_5,V_4)：6 (V_5,V_5)：0	(V_5,V_3)：2
V_3	(V_3,V_0)：0 (V_3,V_1)：5 (V_3,V_2)：0 (V_3,V_3)：0 (V_3,V_4)：6 (V_3,V_5)：0	(V_3,V_1)：5
V_1	(V_1,V_0)：0 (V_1,V_1)：0 (V_1,V_2)：0 (V_1,V_3)：0 (V_1,V_4)：3 (V_1,V_5)：0	(V_1,V_4)：3

因为 Prim()算法中有两重 for 循环，所以时间复杂度为 $O(n^2)$，其中 n 为图中的顶点个数。


视频讲解

7.4.2　克鲁斯卡尔算法

克鲁斯卡尔(Kruskal)算法是一种按权值的递增次序选择合适的边构造最小生成树的方法(选边法)。

基本思路如下。

假设 G＝(V，E)是一个具有 n 个顶点的带权连通无向图，T＝(U，TE)是 G 的最小生成树，则构成最小生成树的步骤如下。

step1：置 U 的初值等于 V(即包含 G 中的全部顶点)，TE 的初值为空集(即图 T 中的每一个顶点都构成一个分量)。

step2：将图 G 中的边按权值从小到大的顺序依次选取，若选取的边未使生成树 T 形成回路，则加入 TE，否则舍弃，直到 TE 中包含(n－1)条边为止。

计算机科学家简介：

Joseph Bernard Kruskal(1928—2010 年)，美国数学家，统计学家和计算机科学家，1954 年获得普林斯顿大学博士学位。当克鲁斯卡尔还是二年级的研究生时，他发明了产生最小生成树的算法，当时他甚至不能肯定关于这个题目的两页半的论文是否值得发表。除了最小生成树外，克鲁斯卡尔还因对多维分析的贡献而著名。

实现 Kruskal 算法的关键是判断选取的边是否与生成树中已保留的边形成回路，这可通过判断边的两个顶点之间是否连通实现。数组 vset[i](初值为 i)代表编号为 i 的顶点所属的连通子图的编号(当选中两个不连通的顶点时，它们分属的两个顶点集合按其中的一个重新统一编号)。当两个顶点的编号不同时，加入这两个顶点构成的边到最小生成树中一定不会形成回路。

在实现 Kruskal 算法 Kruskal()时,用一个数组 E[ ]存放图 G 中的所有边,并且要求它们是按权值从小到大的顺序排列的,为此,先从图 G 的邻接矩阵中获取边集 E,再用直接插入排序法对边集 E 按权值递增排序。Kruskal 算法如下。

```
typedef struct
{ int u;                          //边的起始顶点
 int v;                           //边的终止顶点
 int w;                           //边的权值
}Edge;
void Kruskal(MGraph  * G)
{ int i,j,u1,v1,sn1,sn2,k;
 int vset[MAVX];
 Edge E[MaxSize];                 //存放所有边
 k=0;                             //e 数组的下标从 0 开始
 for(i=0;i<g.n;i++)               //由 g 产生的边集
  for(j=0;j<g.n;j++)
  if(g.edges[i][j]!=0&&g.edges[i][j]!=INF)
  { E[k].u=i;E[k].v=j;E[k].w=g.edges[i][j];
    k++;
    }
insertSort(E,g.e);                //采用直接插入排序对 E 数组按权值递增排序
for(i=0;i<g.n;i++)                //初始化辅助数组
   vset[i]=i;
k=1;                              //k 表示当前构造生成树的第几条边,初值为 1
j=0;                              //E 中边的下标,初值为 0
while(k<g.n)                      //生成的边数小于 n 时循环
{ u1=E[j].u;v1=E[j].v;            //取一条边的头、尾顶点
 sn1=vset[u1];
 sn2=vset[v1];                    //分别得到两个顶点所属的集合编号
 if(sn1!=sn2)                     //两顶点属于不同的集合,该边是最小生成树的一条边
 { printf("(%d,%d):%d\n",u1,v1,E[j].w);
   k++;                           //生成边数增 1
   for(i=0;i<g.n;i++)             //两个集合统一编号
    if(vset[i]==sn2)              //集合编号为 sn2 的改为 sn1
     vset[i]=sn1;
}
 j++;                             //扫描下一条边
 }
}
```

如果给定的带权连通无向图 G 有 n 个顶点、e 条边,在上述算法中,对边集 E 采用直接插入排序的时间复杂度为 $O(e^2)$。While 循环是在 e 条边中选取(n−1)条边,最坏情况下执行 e 次,而其中的 for 循环执行 n 次,因此,while 循环的时间复杂度为 $O(n^2+e)$。对于连通无向图,$e \geq (n-1)$,那么,用 Kruskal 算法构造最小生成树的时间复杂度为 $O(e^2)$。

例如,图 7.23 中的带权无向图调用 Kruskal 算法 Kruskal(g)求解最小生成树的过程如图 7.24 所示。其中,数组 E 排序(按边的权值从小到大排序,每个边的起点为编号较小的顶点,终点为编号较大的顶点)后的结果如下。

{V_0,V_2,1},{V_2,V_0,1},{V_3,V_5,2},{V_5,V_3,2},{V_1,V_4,3},{V_4,V_1,3},{V_2,V_5,4},{V_5,V_2,4},{V_1,V_2,5},{V_2,V_1,5},{V_0,V_3,5},{V_3,V_0,5},{V_2,V_3,5},{V_3,V_2,5},{V_0,V_1,6},{V_1,V_0,6},{V_2,V_4,6},{V_4,V_2,6},{V_4,V_5,6},{V_5,V_4,6};

初始时,顶点 i 对应的 vset[i]值为 i,图 7.24 中各顶点旁边标出了该值的变化过程。在图 7.24 标号①中生成一条边＜V_0,V_2＞,顶点 V_0 和 V_2 连通,则将顶点 V_2 的 vset[2]的值改为 0。在图 7.24 标号②中生成一条边＜V_3,V_5＞,顶点 V_3 和 V_5 连通,则将顶点 V_5 的 vset[5]的值改为 3。在图 7.24 标号③中生成一条边＜V_1,V_4＞,顶点 V_1 和 V_4 连通,则将顶点 V_4 的 vset[4]的值改为 1。在图 7.24 标号④中生成一条边＜V_2,V_5＞,这样,顶点 V_0,V_2,V_3,V_5 连通,则将顶点 V_5 的 vset[5]的值改为 0。顶点 V_3 的 vset[3]的值改为 0。在图 7.24 标号⑤中增加一条边＜V_1,V_2＞,这样,所有顶点都连通,则将顶点 V_1 和 V_4 外的所有顶点 i 的 vset[i]值改为 1。

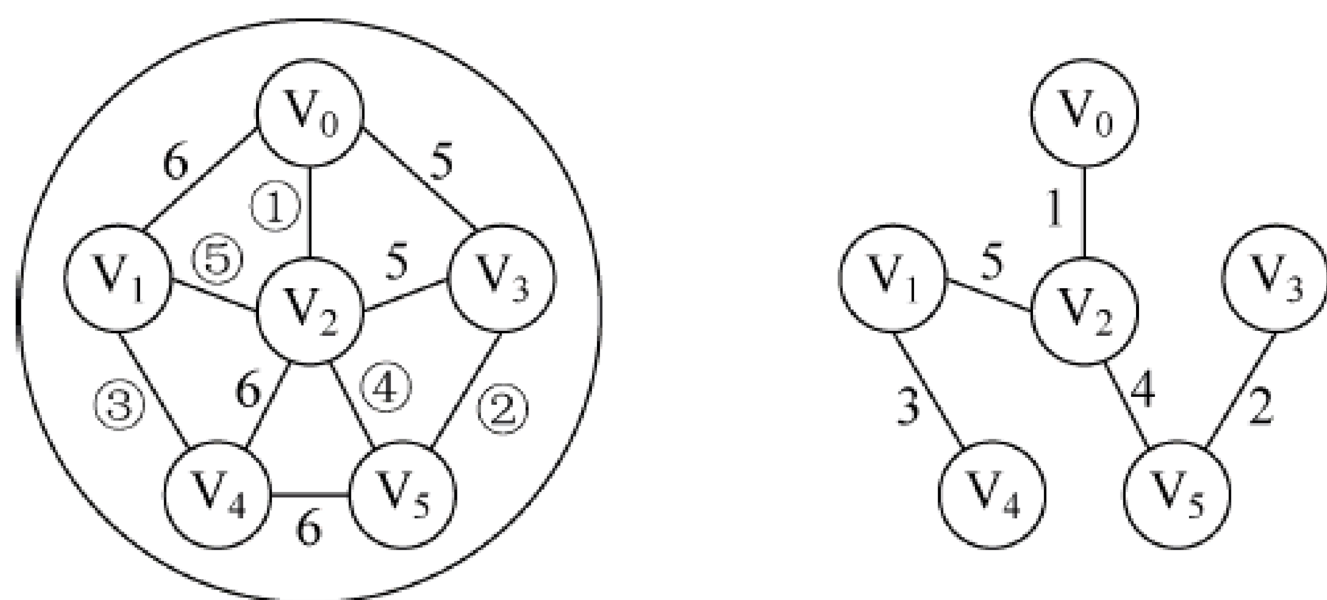


图 7.24　图 7.23 中的带权无向图 Kruskal 算法求解最小生成树的过程

可以对前面的 Kruskal 算法进行两方面的改进:其一是将边集排序改为堆排序(将在后面介绍);其二是采用之前介绍的并查集进行顶点合并,先通过 MAKE_SET(t,n)进行并查集树的初始化,即每个顶点作为一个分离集合树(其编号为该顶点的编号),当找到一条最小边(u,v)时,求出两者所在分离集合树的编号,若不同,则将顶点 u 和顶点 v 所在的分离集合树按秩合并,对应的算法如下。

```
void MAKE_SET(UFSTree t[ ],int n)                 //初始化并查集树
{ int i;
 for(i=0;i<n;i++)                                 //顶点编号为 0~(n-1)
 { t[i].rank=0;                                   //置初始化为 0
   t[i].parent=i;                                 //双亲初始化指向自己
 }
}
int FIND_SET(UFSTree t[ ],int x)                  //在 x 所在子树中查找集合编号
{ if(x!=t[x].parent)                              //若双亲不是自己
     return(FIND_SET(t,t[x].parent));             //递归在双亲中找 x
else
     return(x);                                   //若双亲是自己,则返回 x
}
void UNION(UFSTree t[ ],int x ,int y)             //将 x 和 y 所在的子树合并
{ x=FIND_SET(t,x);
 y=FIND_SET(t,y);
 if(t[x].rank>t[y].rank)                          //y 结点的秩小于等于 x 结点的秩
      t[y].parent=x;                              //将 y 连到 x 结点上,x 作为 y 的孩子结点
```

```
    else                                  //y 结点的秩大于等于 x 结点的秩
    { t[x].parent=y;                      //将 x 连到 y 结点上,y 作为 x 的孩子结点
      if(t[x].rank==t[y].rank)            //x 和 y 结点的秩相同
         t[y].rank++;                     //y 结点的秩增 1
    }
}
void Kruskal(MGraph g)
{ int i,j,u1,v1,sn1,sn2,k;
  UFSTree t[MaxSize];
  Edge E[MaxSize];                        //存放所有边
  k=1;                                    //e 数组的下标从 1 开始
  for(i=0;i<g.n;i++)                      //由 g 产生的边集 E
   for(j=0;j<g.n;j++)
    if(g.edges[i][j]!=0&&g.edges[i][j]!=INF)
    { E[k].u=i;E[k].v=j;E[k].w=g.edges[i][j];
      k++;
    }
  HeapSort(E,g.e);                        //采用堆排序对 E 数组按权值递增排序
  MAKE_SET(t,g.n);                        //初始化并查集树 t
  k=1;                                    //k 表示当前构造生成树的第几条边,初值为 1
  j=1;                                    //E 中边的下标,初值为 1
  while(k<g.n)                            //生成的边数小于 n 时循环
  { u1=E[j].u;v1=E[j].v;                  //取一条边的头、尾顶点,编号为 u1 和 v1
    sn1=FIND_SET(t,u1);
    sn2=FIND_SET(t,v1);                   //分别得到两个顶点所属的集合编号
    if(sn1!=sn2)                          //两顶点属于不同的集合,该边是最小生成树的一条边
    { printf("(%d,%d):%d\n",u1,v1,E[j].w);
      k++;                                //生成边数增 1
      UNION(t,u1,v1);
    }
    j++;                                  //扫描下一条边
  }
}
```

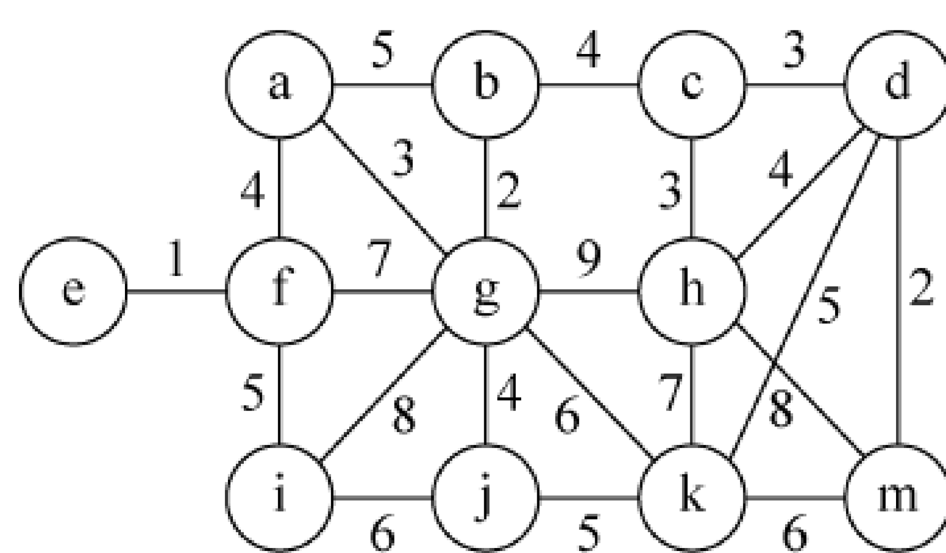
如果给定的带权连通无向图 G 有 n 个顶点、e 条边,上述改进的 Kruskal 算法中,不考虑生成边数组 E 的过程,堆排序的时间复杂度为 $O(e\log_2 e)$。while 循环是在 e 条边中选取(n−1)条边,最坏情况下执行 e 次,而其中的 UNION()的执行时间为 $O(\log_2 n)$。对于连通无向图,$e \geq n-1$,那么,改进的 Kruskal 算法构造最小生成树的时间复杂度为 $O(e\log_2 e)$。不作特殊说明,通常认为 Kruskal 算法的时间复杂度为 $O(e\log_2 e)$。

【例 7.7】 用 Kruskal 算法求如图 7.25 所示的赋权图的最小生成树。

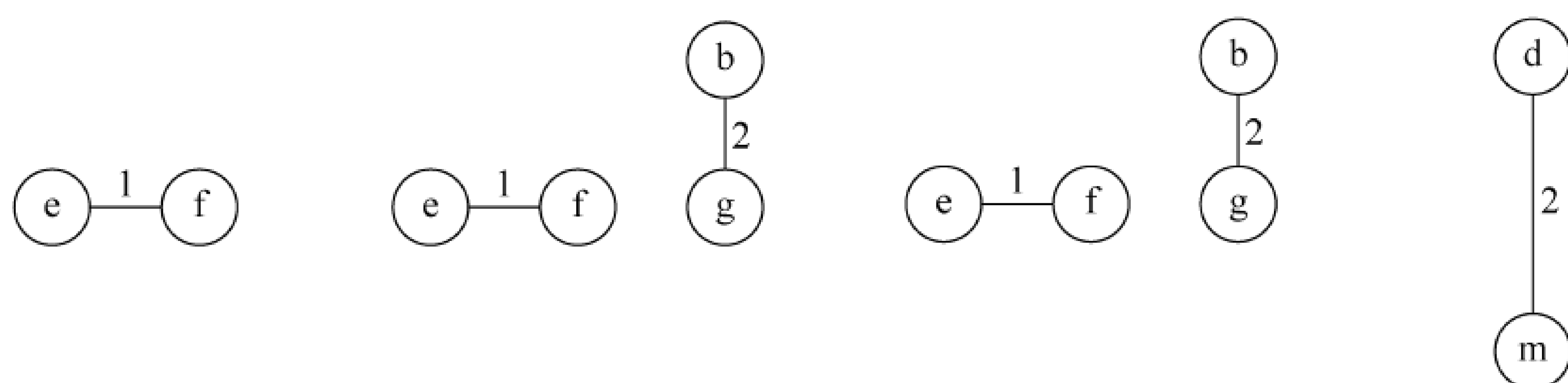
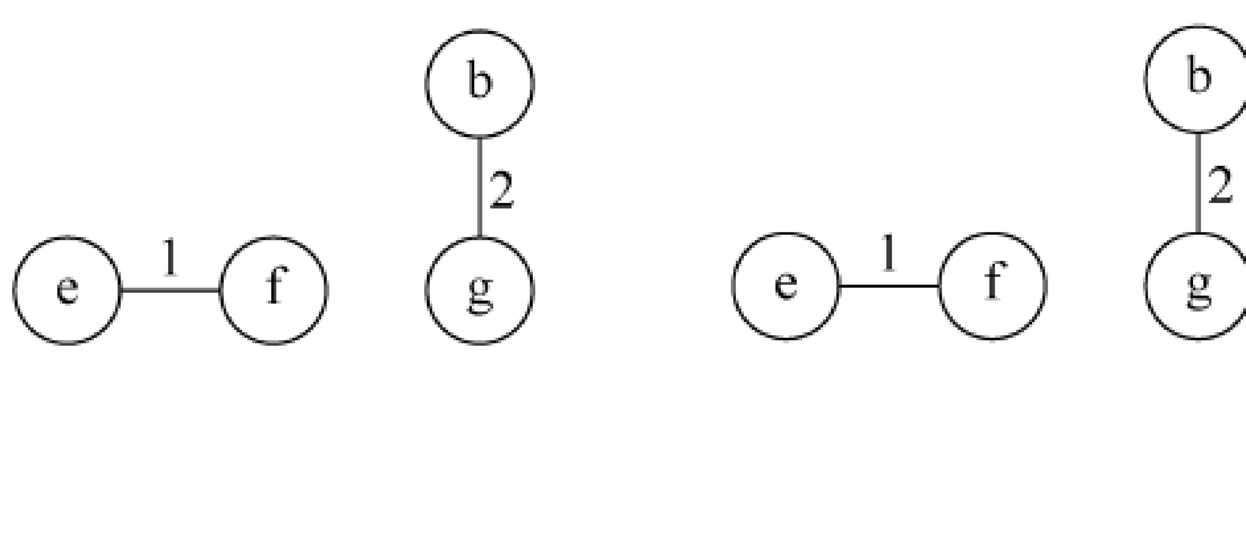
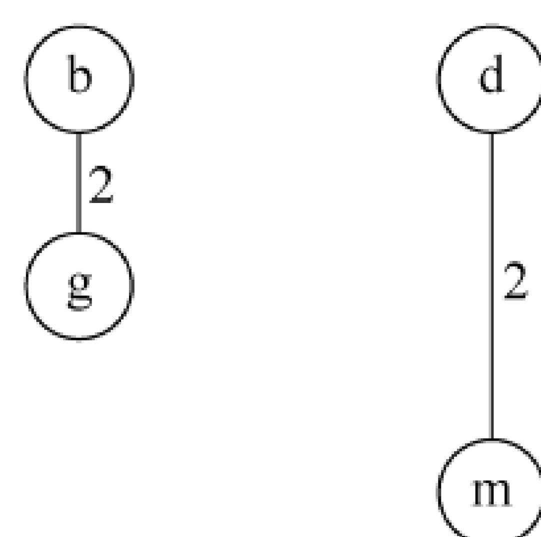
注意:因为图 7.25 中的顶点个数 n=12,所以按算法要执行 n−1=11 次,其过程如图 7.25(a)~(k)所示。

【例 7.8】 用 Prim 算法求如图 7.26 赋权图的最小生成树(假设已知顶点 a)。

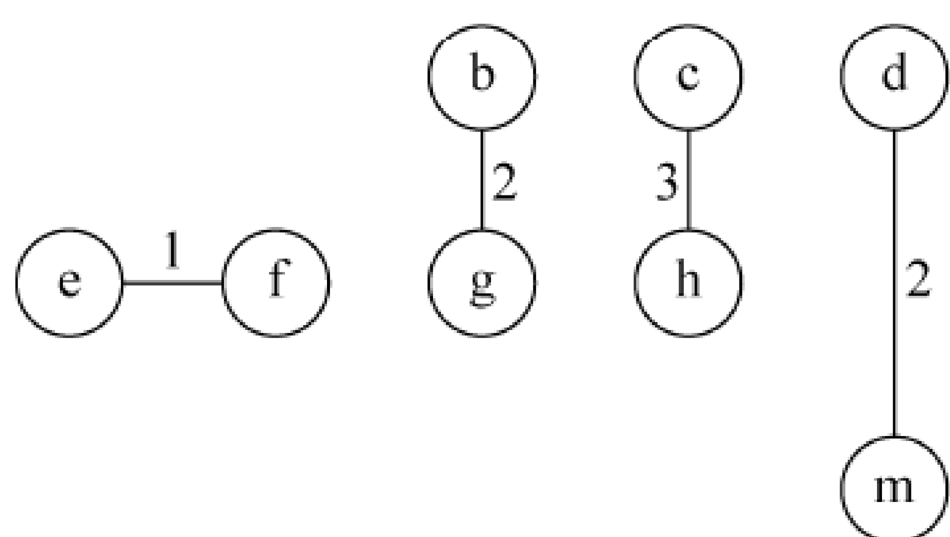
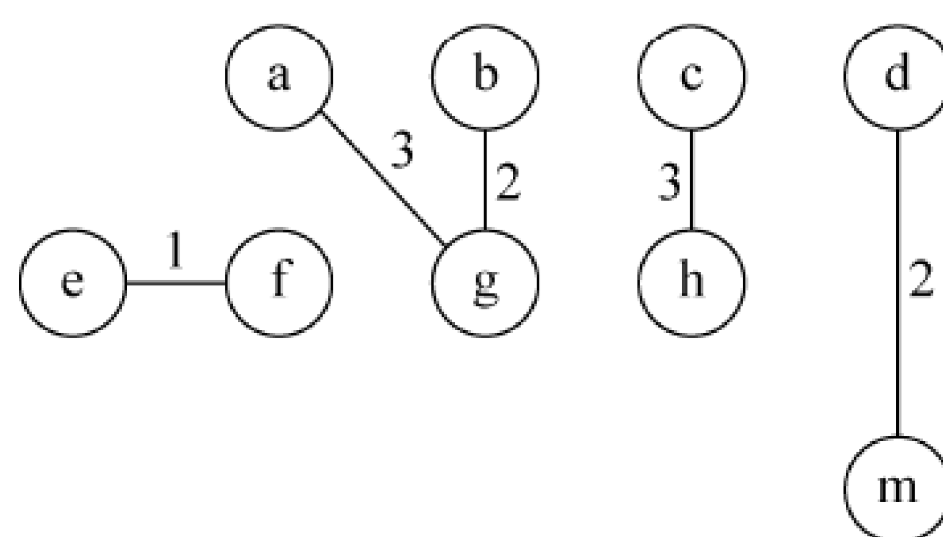
注意:因为图的顶点个数 n=7,所以按照算法要执行 n−1=6 次。由 Prim 算法可以看出,每一步得到的图一定是树,因此不需要验证是否包含回路,它的计算工作量较 Kruskal 算法要小。



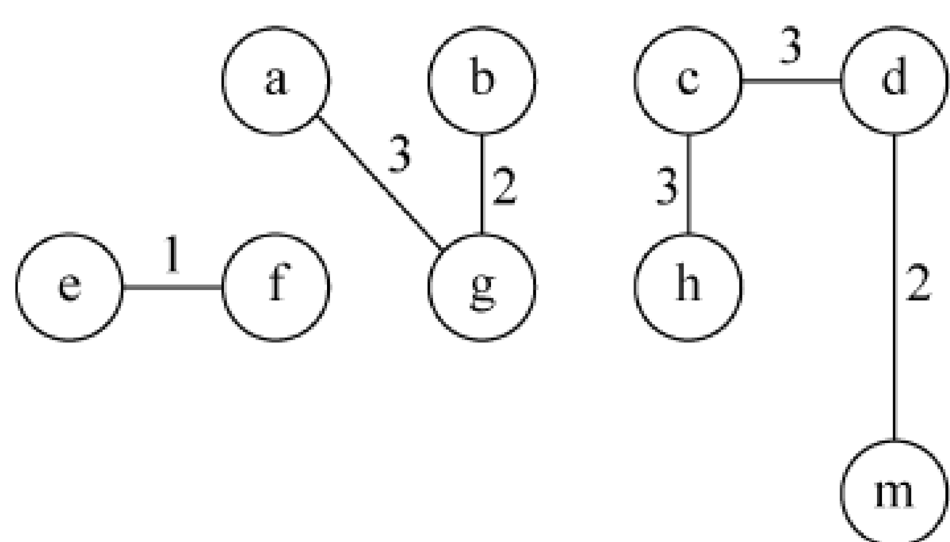
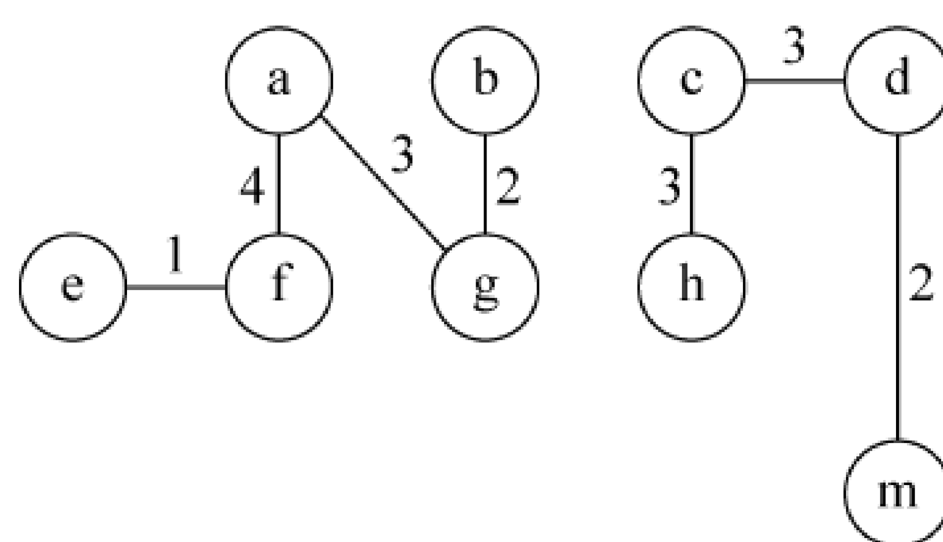
赋权图

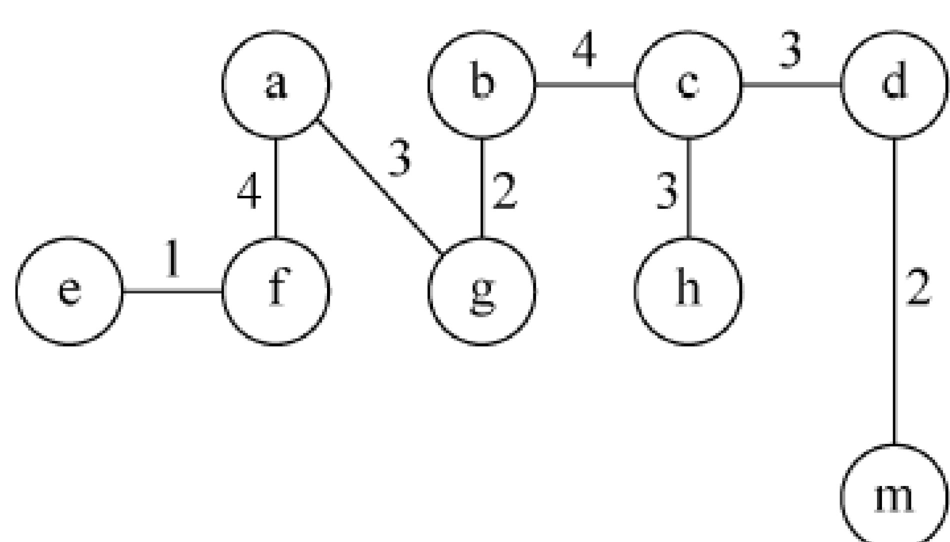
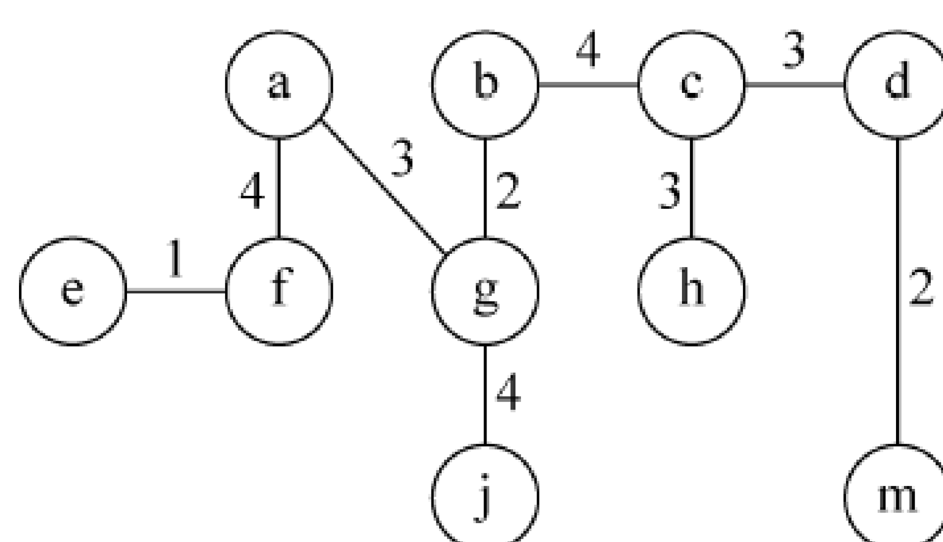
(a) 选权为1的边　　(b) 选权为2的边1　　(c) 选权为2的边2

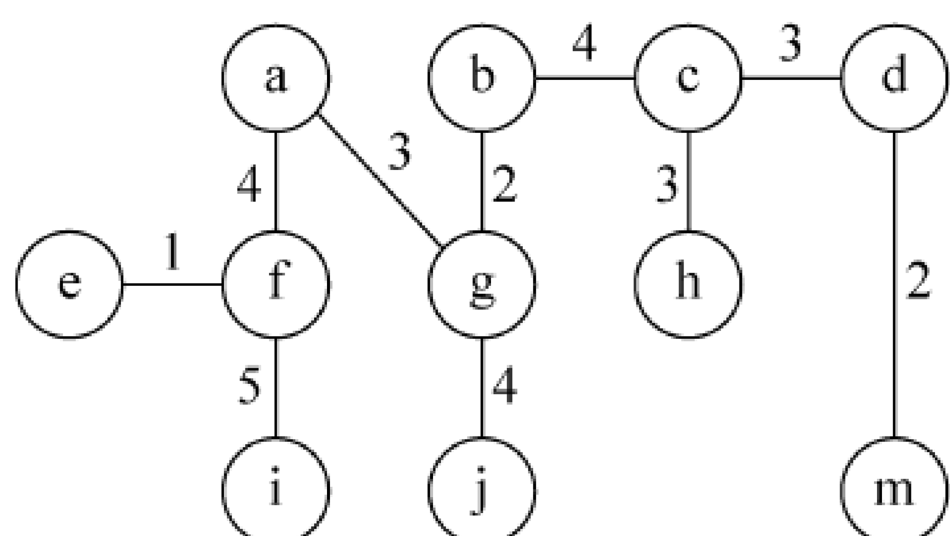
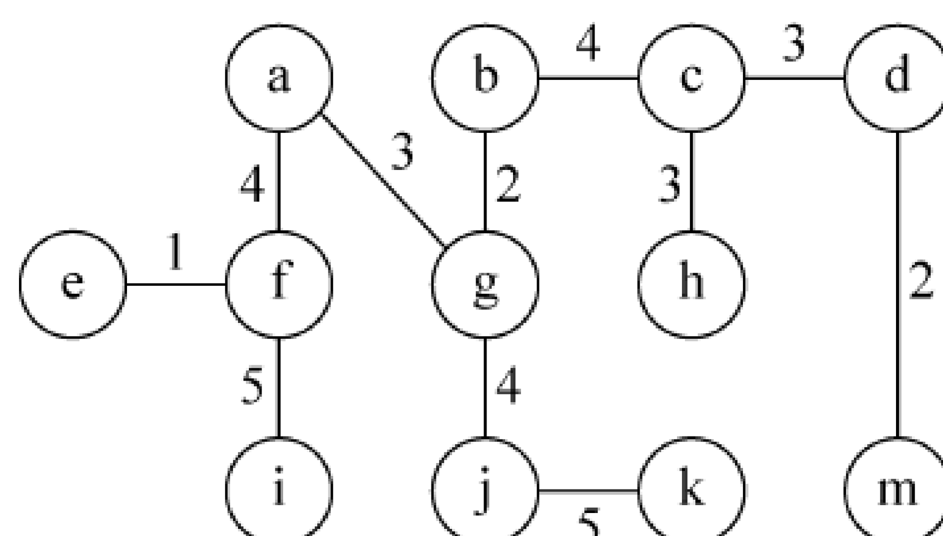
(d) 选权为3的边1　　(e) 选权为3的边2

(f) 选权为3的边3　　(g) 选权为4的边1

(h) 选权为4的边2　　(i) 选权为4的边3

(j) 选权为5的边1　　(k) 选权为5的边2

图 7.25　Kruskal 算法求最小生成树

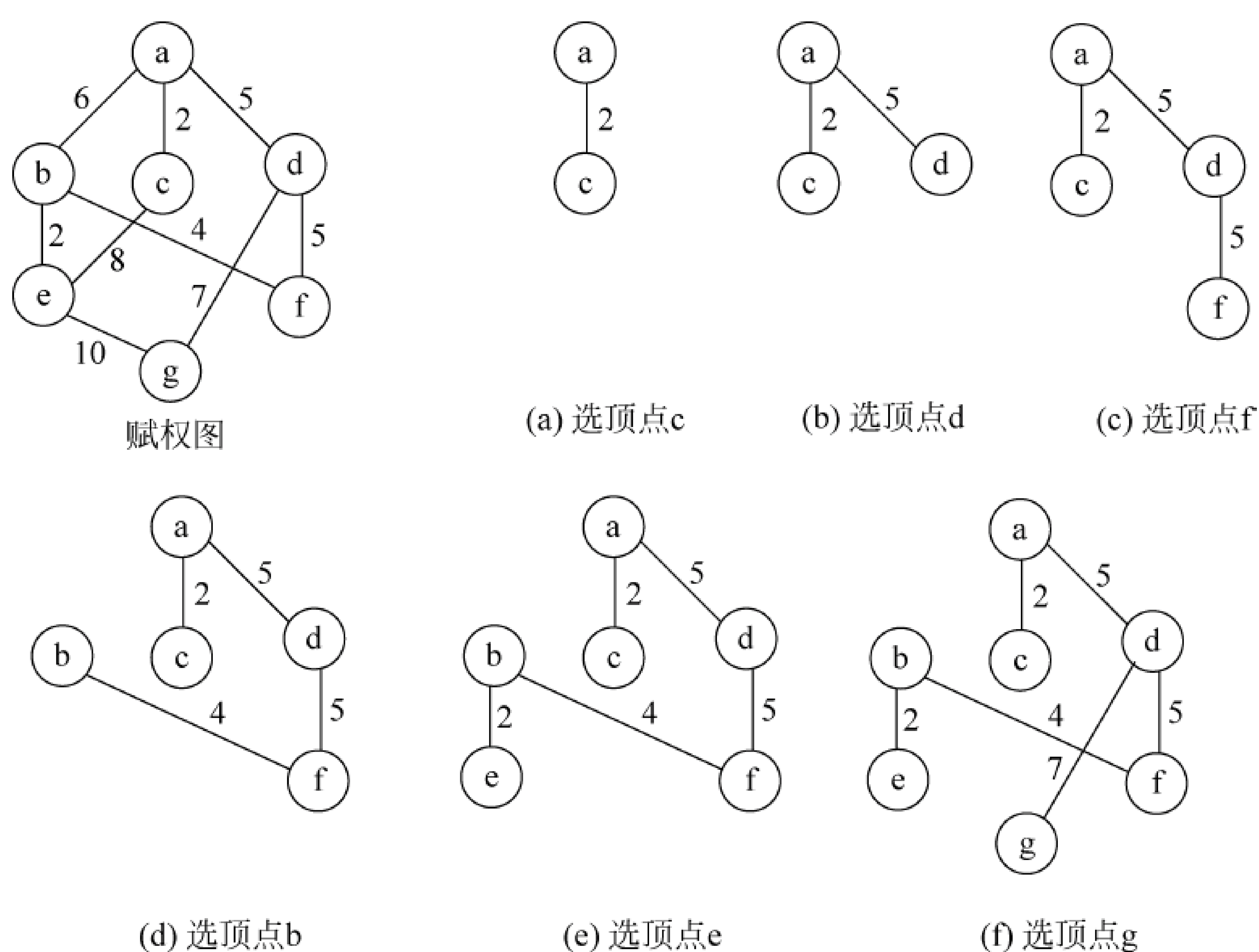


图 7.26　Prim 算法求最小生成树

【例 7.9】 假设有 5 个信息中心 A,B,C,D,E,它们之间的距离如图 7.27 所示。要交换数据,可以在任意两个信息中心之间通过光纤等网络介质连接,但是,由于费用的局限,要求铺设尽可能少的光纤线路。可以通过其他中心转发。

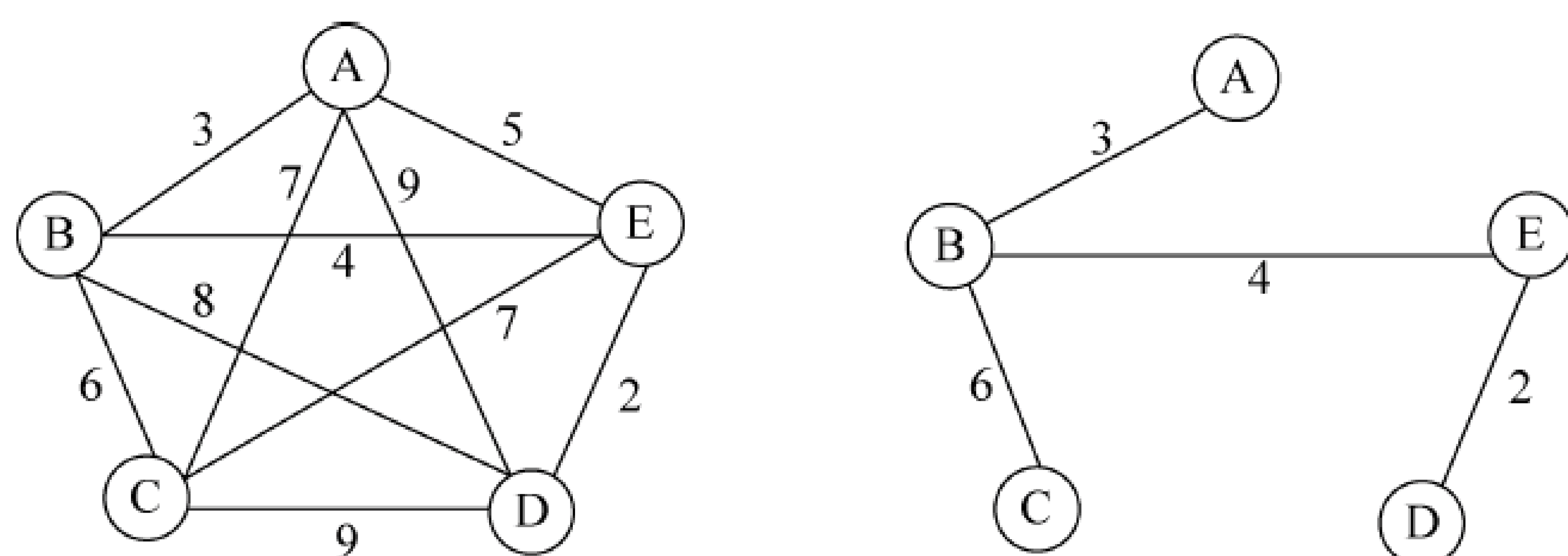


图 7.27　信息中心网络图及最小生成树

分析:这实际上就是求赋权连通图的最小生成树问题,可以通过 Prim 算法或 Kruskal 算法求解。求解结果如上图所示,即按右图铺设线路最短,总长度 L=15。

7.5　最短路径

在一个无权图中,若一个顶点到另一个顶点存在一条路径,则该路径长度为该路径上所经过的边的数目,等于该路径上的顶点数减 1。从一个顶点到另一个顶点可能存在多条路径,每条路径上所经过的边数可能不同,即路径长度不同,把路径长度最短(即经过的边数最少)的那条路径叫作**最短路径**,其路径长度叫作**最短路径长度**或**最短距离**。

对于带权的图,应考虑路径上各边的权值,通常把一条路径上经过的边的权值之和定义为该路径的**路径长度**或**带权路径长度**。从源点到终点可能不止一条路径,把带权路径长度

最短的那条路径称为**最短路径**，其路径长度（权值之和）称为**最短路径长度**或**最短距离**。

实际上，只要把无权图上的每条边看成是权值为 1 的边，那么无权图和带权图的最短路径和最短距离的定义就是一致的。

求图的最短路径的问题分为两个方面：单源最短路径（即求图中一个顶点到其余各顶点的最短路径）和全源最短路径（即求图中每对顶点之间的最短路径）。

7.5.1 单源最短路径


视频讲解

从某源点到其余各顶点的最短路径：从给定顶点（单源点）求到图中其他各顶点的最短路径。

问题：给定一个带权有向图 G 与源点 v，求从顶点 v 到 G 中其他顶点的最短路径，并限定各边的权值大于或等于 0。

采用迪杰斯特拉（Dijkstra）算法求解，其基本思想如下。

首先，设 G＝（V，E）是一个带权有向图，把图中的顶点集合 V 分成两组：

第一组，已求出最短路径的顶点集合（用 S 表示，初始时 S 只有一个源点）；

第二组，其余未确定最短路径的顶点集合（用 U 表示）。

其次，求下一条最短路径：

step1：$V_i \in V-S$，先求出 V_0 到 V_i 中间只经过在 S 中结点的最短路径。

step2：上述集合 U 中最短路径长度最小者即为下一条最短路径；将其路径终点加入 S 中。

step3：重复 **step2**，直到所有顶点都加入 S 中。

在向 S 中添加顶点时，总保持从源点 v 到 S 中各顶点的最短路径长度不大于从源点 v 到 U 中任何顶点的最短路径长度。例如，若向 S 中添加的是顶点 u，对于 U 中的每个顶点 j，如果顶点 u 到顶点 j 有边（权值为 w_{uj}），且原来从顶点 v 到顶点 j 的路径长度（c_{vj}）大于从顶点 v 到顶点 u 的路径长度（c_{vu}）与 w_{uj} 之和，即 $c_{vj}>c_{vu}+w_{uj}$，如图 7.28 所示，则将 v＝＞u＝＞j 的路径作为新的最短路径。

实际上，从顶点 v 到顶点 j 的这条最短路径是只包括 S 中的顶点为中间点的当前最短路径长度，随着 S 中的顶点不断增加，当 S 包含所有顶点时，这条新的最短路径就是最终的最短路径。

Dijkstra 算法的具体步骤如下。

step1：开始时，S 只包含源点 v，即 S＝{v}，顶点 v 到自己的距离为 0。U 包含除 v 外的其余顶点，v 到 U 中顶点 i 的距离为边上的权（若 v 与 i 有边＜v，i＞）或∞（若 i 不是 v 的出边邻接点）。

step2：从 U 中选取一个顶点 u，顶点 v 到顶点 u 的距离最小，然后把顶点 u 加到 S 中（该选定的距离就是 v 到 u 的最短路径长度）。

step3：以顶点 u 为新考虑的中间点，修改顶点 v 到 U 中各顶点的距离。若从源点 v 到顶点 j（j∈U）经过顶点 u 的距离（图 7.28 中的 $c_{vu}+w_{uj}$）比原来不经过顶点 u 的距离（图 7.28 中 c_{vj}）短，则修改从顶点 v 到顶点 j 的最短距离值（图 7.28 中修改为 $c_{vu}+w_{uj}$）。

step4：重复步骤 **step2** 和 **step3**，直到 S 包含所有顶点。

例如，对于图 7.29 所示的带权有向图，采用 Dijkstra 算法求从顶点 0 到其他顶点的最短路径时，S、U 和从 v（这里 v 等于 0，即源点编号）到各顶点的距离的变化如下（S 中加 * 号

者表示新加入的点,距离中加 * 号者表示修改后的距离值)。

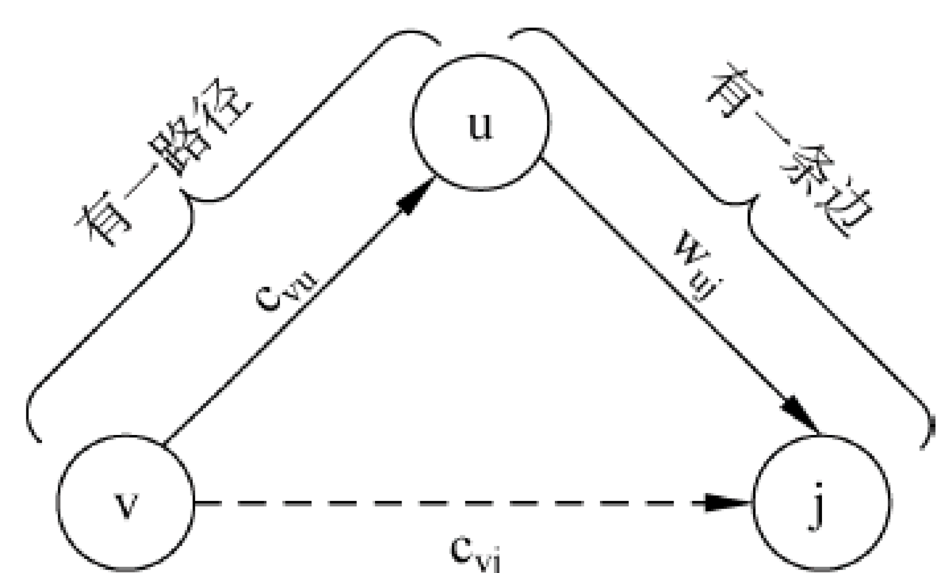


图 7.28　从顶点 v 到顶点 j 的路径比较

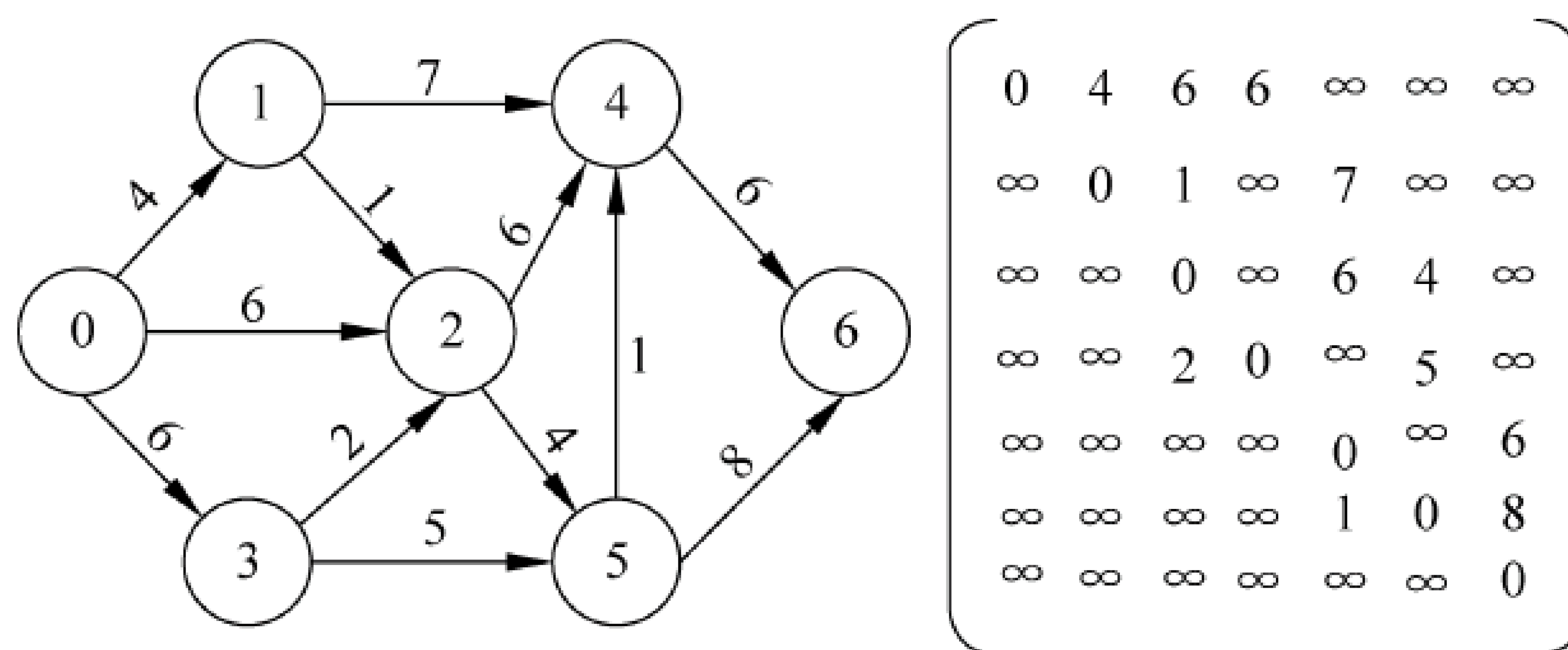


图 7.29　一个有向图及其邻接矩阵

S	U	v 到 0～6 各顶点的距离
{0}	{1,2,3,4,5,6}	{0,4,6,6,∞,∞,∞}
{0,1*}	{2,3,4,5,6}	{0,4,5*,6,11*,∞,∞}
{0,1,2*}	{3,4,5,6}	{0,4,5,6,11,9*,∞}
{0,1,2,3*}	{4,5,6}	{0,4,5,6,11,9,∞}
{0,1,2,3,5*}	{4,6}	{0,4,5,6,10*,9,17*}
{0,1,2,3,5,4*}	{6}	{0,4,5,6,10,9,16*}
{0,1,2,3,5,4,6}	{}	{0,4,5,6,10,9,16}

顶点 0 到 1～6 各顶点的最短距离分别为 4,5,6,10,9,16。

下面介绍 Dijkstra 算法的实现过程。设有向图 G=(V,E),以邻接矩阵 g 作为存储结构,并规定:

$$g.edges[i][j]=\begin{cases} w_{ij} & \text{当 } i\neq j \text{ 且}(i,j)\in E(G) \\ 0 & \text{当 } i=j \\ \infty & \text{其他} \end{cases}$$

设置一维数组 s[0..n−1],用于标记已找到最短路径的顶点,并规定:

$$s[i]=\begin{cases} 0 & \text{未找到源点到顶点 } i \text{ 的最短路径} \\ 1 & \text{已找到源点到顶点 } i \text{ 的最短路径} \end{cases}$$

设置数组 dist[0..n−1],dist[i]用来保存从源点 v 到顶点 i 的目前最短路径长度,它的初值为<v,i>边上的权值,若顶点 v 到顶点 i 没有边,则权值定为∞。以后每考虑一个新的中间点时,dist[i]的值可能被修改变小。

另设置一个数组 path[ ]用于保存最短路径长度。如图 7.28 所示,若顶点 v 到顶点 u 是最短路径,而顶点 u 到顶点 j 有一条边,则顶点 v 到顶点 j 的最短路径为顶点 v 到顶点 u 的最短路径+顶点 u 到顶点 j 的边长。所以,只用 path[j]保存 u,再由 path[u]一步一步向前推,直到源点 v,这样可以推出从源点 v 到顶点 j 的最短路径。也就是说,path[j]保存当前最短路径中的前一个顶点的编号,它的初值为源点 v 的编号(顶点 v 到顶点 i 有边时)或−1(顶点 v 到顶点 i 无边时)。

Dijkstra 算法如下(n 为图 G 的顶点数,v 为源点编号)。

```
void Dijkstra(MGraph g,int v)
{ int dist[MAVX],path[MAVX];
 int s[MAVX];
 int mindis,i,j,u;
 for(i=0;i<g.n;i++)
 { dist[i]=g.edges[v][i];                    //距离初始化
   s[i]=0;                                   //s[]值置空
   if(g.edges[v][i]<INF)                     //路径初始化
      path[i]=v;                             //顶点 v 到顶点 i 有边时,置顶点 i 的前一个顶点为 v
   else
      path[i]=-1;                            //顶点 v 到顶点 i 无边时,置顶点 i 的前一个顶点为-1
  }
s[v]=1;                                      //源点编号 v 放入 S 中
path[v]=0;
for(i=0;i<g.n;i++)                           //循环,直到所有顶点的最短路径都求出
{ mindis=INF;                                //mindis 置最小长度初值
 for(j=0;j<g.n;j++)                          //选取不在 s 中且具有最小距离的顶点 u
   if(s[j]==0&&dist[j]<mindis)
 { u=j;
    mindis=dist[j];
 }
s[u]=1;                                      //顶点 u 加入到 S 中
for(j=0;j<g.n;j++)                           //修改不在 s 中的顶点的距离
 if(g.edges[u][j]<INF&&dist[u]+g.edges[u][j]<dist[j])
 { dist[j]=dist[u]+g.edges[u][j];
    path[j]=u;
   }
 }
 Dispath(g,dist,path,s,v);                   //输出最短路径
}
```

通过 path[i]向前推,直到顶点 0 为止,可以找出从顶点 v 到顶点 i 的最短路径。例如,对于顶点 0~顶点 6,计算出 path 如下。

初始值	0	1	初始值	5	2	4

求顶点 0 到顶点 6 的路径计算过程是:path[6]=4,说明路径上顶点 6 之前的一个顶点是 4; path[4]=5,说明路径上顶点 4 之前的一个顶点是 5; path[5]=2,说明路径上顶点 5 之前的一个顶点是 2; path[2]=1,说明路径上顶点 2 之前的一个顶点是 1; path[1]=0,说明路径上顶点 1 之前的一个顶点是 0。因此,顶点 0 到顶点 6 的路径为 0,1,2,5,4,6。

输出最短路径的 Dispath()函数如下。

```
void Dispath(MGraph g,int dist[],int path,int S[],int v)
//输出从顶点 v 出发的所有最短路径
{ int i,j,k;
 int apath[MAVX],d;                          //存放一条最短路径(逆向)及其顶点个数
```

```
    for(i=0;i<g.n;i++)                          //循环输出顶点 v 到顶点 i 的路径
     if(S[i]==1&&i!=v)
      { printf("从顶点%d 到顶点%d 的路径长度为:%d\t",v,i,dist[i]);
        d=0;apath[d]=i;                         //添加路径上的终点
        k=path[i];
        if(k==-1)                               //无路径的情况
          printf("无路径\n");
        else                                    //存在路径时输出该路径
        { while(k!=v)
          { d++;apath[d]=k;
            k=path[k];
          }
          d++;apath[d];                         //添加路径上的起点
          printf("%d",apath[d]);                //先输出起点
          for(j=d-1;j>=0;j--)                   //再输出其他顶点
             printf(",%d",apath[j]);
          printf("\n");
        }
      }
    }
```

Dijkstra 算法的时间复杂度为 $O(n^2)$,其中 n 为图中的顶点个数。

【例 7.10】 对如图 7.29 所示的有向图,采用 Dijkstra 算法求从顶点 0 到其他顶点的最短路径,并说明整个计算过程。

解:

(1) 初值:s[ ]={0},dist[ ]={0,4,6,6,∞,∞,∞}(顶点 0 到其他顶点的权值),path[ ]={0,0,0,0, −1,−1,−1}(顶点 0 到其他顶点有路径时为 0,否则为−1)。

(2) 从 dist[ ]中找除 s[ ]中顶点外最近的顶点 1,加入 s 中,s[ ]={0,1},从顶点 1 到达顶点 2 和顶点 4 有边:

dist[2]=min{dist[2],dist[1]+1}=5(修改)

dist[4]=min{dist[4],dist[1]+7}=11(修改)

则 dist[ ]={0,4,5,6,11, ∞,∞},用顶点 1 替换修改 dist 值的顶点,path[ ]={0,0,1,0,1,−1,−1}。

(3) 从 dist[ ]中找除 s[ ]中顶点外最近的顶点 2,加入 s 中,s[ ]={0,1,2},从顶点 2 到达顶点 4 和顶点 5 有边:

dist[4]=min{dist[4],dist[2]+6}=11

dist[5]=min{dist[5],dist[2]+4}=9(修改)

则 dist[ ]={0,4,5,6,11,9,∞},用顶点 2 替换修改 dist 值的顶点,path[ ]={0,0,1,0,1,2,−1}。

(4) 从 dist[ ]中找除 s[ ]中顶点外最近的顶点 3,加入 s 中,s[ ]={0,1,2,3},从顶点 3 到达顶点 2 和顶点 5 有边:

dist[2]=min{dist[2],dist[3]+2}=5

dist[5]=min{dist[5],dist[3]+5}=9

没有修改，dist[ ]和 path[ ]不变。

(5) 从 dist[ ]中找除 s[ ]中顶点外最近的顶点 5，加入 s 中，s[ ]={0,1,2,3,5}，从顶点 5 到达顶点 4 和顶点 6 有边：

dist[4]=min{dist[4],dist[5]+1}=10(修改)

dist[6]=min{dist[6],dist[5]+8}=17(修改)

则 dist[ ]={0,4,5,6,10,9,17}，将 5 替换修改 dist 值的顶点，path[ ]={0,0,1,0,5,2,5}。

(6) 从 dist[ ]中找除 s[ ]中顶点外最近的顶点 4，加入 s 中，s[ ]={0,1,2,3,5,4}，从顶点 4 到达顶点 6 有边：

dist[6]=min{dist[6],dist[4]+6}=16(修改)

则 dist[ ]={0,4,5,6,11,9,16}，将 5 替换修改 dist 值的顶点，path[ ]={0,0,1,0,5,2,4}。

(7) 从 dist[ ]中找除 s[ ]中顶点外最近的顶点 6，加入 s 中，s[ ]={0,1,2,3,5,4,6}，从顶点 6 不能到达任何顶点。算法结束，此时 dist[ ]={0,4,5,6,10,9,16}，path[ ]={0,0,1,0,5,2,4}。

本算法的求解过程如下。

从顶点 0 到顶点 1 的路径长度为 4，路径为 0,1。

从顶点 0 到顶点 2 的路径长度为 5，路径为 0,1,2。

从顶点 0 到顶点 3 的路径长度为 6，路径为 0,3。

从顶点 0 到顶点 4 的路径长度为 10，路径为 0,1,2,5,4。

从顶点 0 到顶点 5 的路径长度为 9，路径为 0,1,2,5。

从顶点 0 到顶点 6 的路径长度为 16，路径为 0,1,2,5,4,6。

Dijkstra 算法具有如下特点。

- 不适合含有负权值的带权图求单源最短路径。下面通过一个反例说明。假设一个含有 3 个顶点的带权有向图，<0,1>的权值为 1，<0,2>的权值为 2，<2,1>的权值为−3。若源点为 0，在执行 Dijkstra 算法时，首先选取的中间点为 1，以后不再改变。实际上，0→2→1 才是源点 0 到顶点 1 的最长路径，其长度为−1。
- 假设在算法执行中添加到 S 中的顶点顺序是 $u_1,u_2,\cdots,u_m$，则源点 v 到 u_1、源点 v 到 u_2……源点 v 到 u_m 的最短路径长度是递增的。也就是说，源点 v 到 u_2 的最短路径长度一定大于源点 v 到 u_1 的最短路径长度，以此类推。
- 一旦某个顶点 u 添加到 S 中(表示已经求出源点 v 到该顶点的最短路径长度)，则在算法后面的执行中，不会再修改源点 v 到顶点 u 的最短路径长度。

分析这个算法的运行时间，时间复杂度为 $O(n^2)$。

【例 7.11】 求如图 7.30 所示的无向赋权图中顶点 V_1 到 V_6 的最短通路。

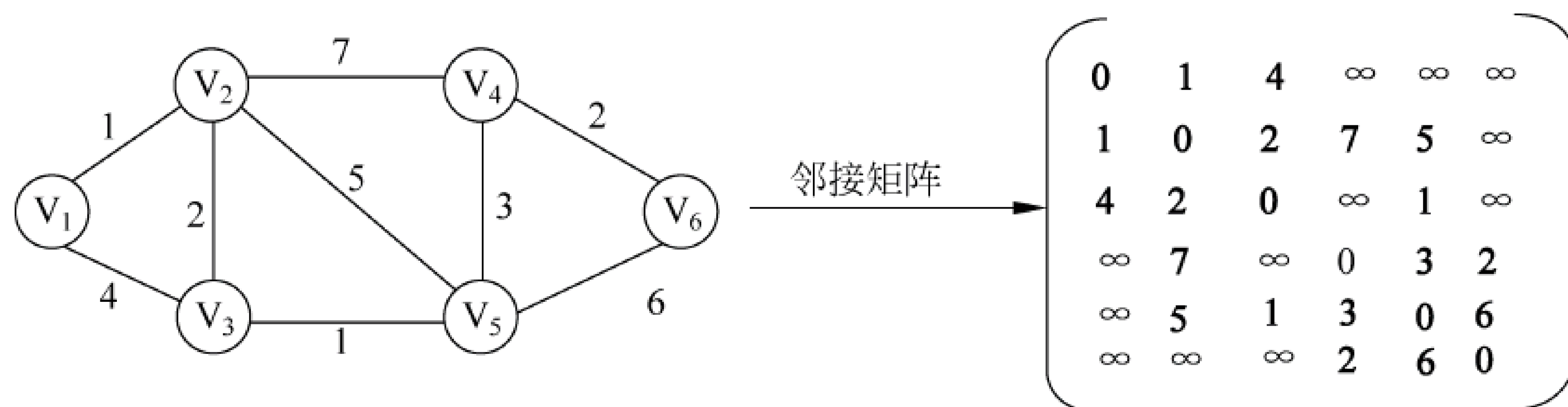
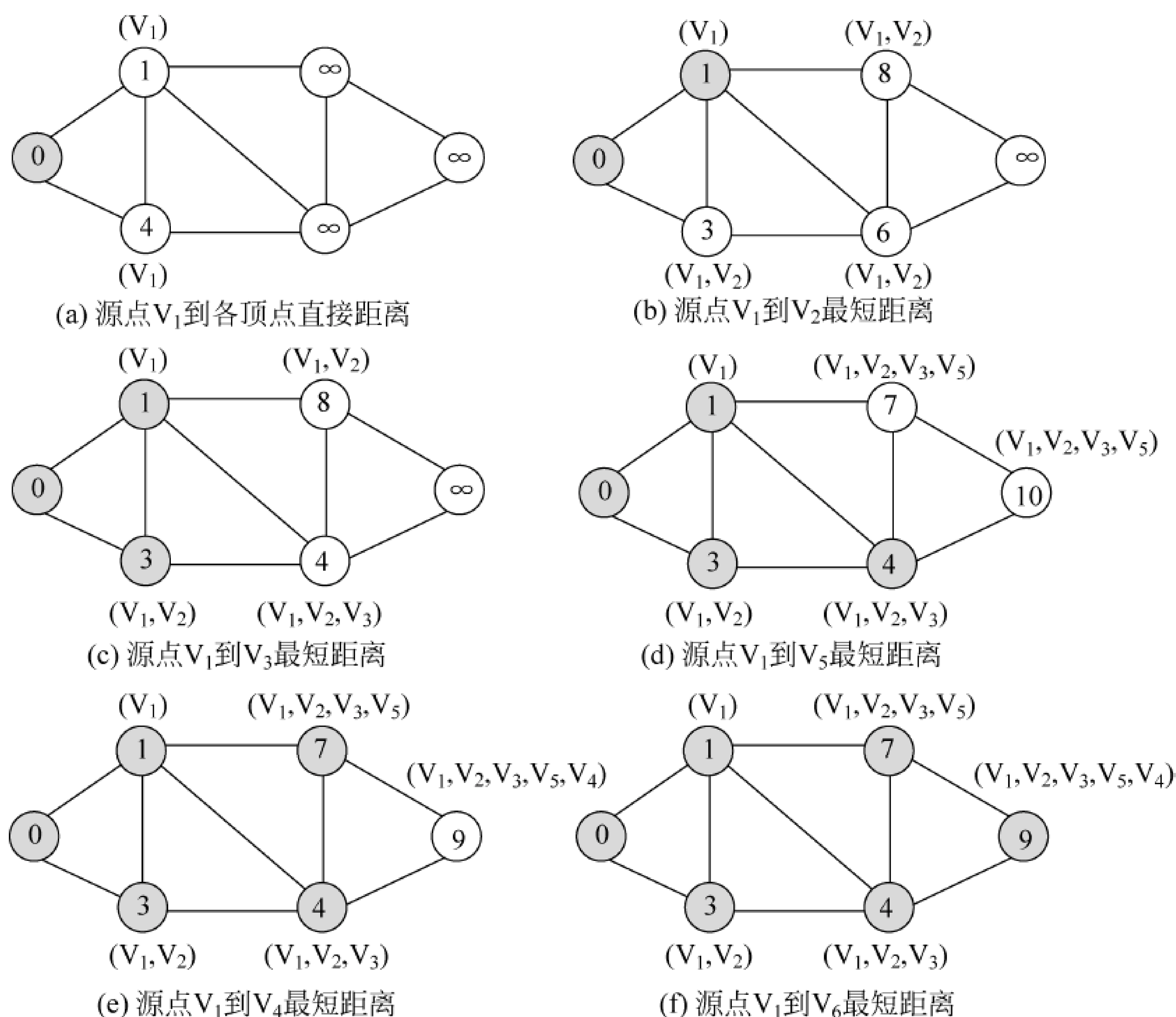


图 7.30 无向赋权图及邻接矩阵

解：最短路径的求解过程如图 7.31 所示。



终点	i=1	i=2	i=3	i=4	i=5
V_2	1 (V_1,V_2)				
V_3	4 (V_1,V_3)	3 (V_1,V_2,V_3)			
V_4	∞	8 (V_1,V_2,V_4)	8 (V_1,V_2,V_4)	7 (V_1,V_2,V_3,V_5,V_4)	
V_5	∞	6 (V_1,V_2,V_5)	4 (V_1,V_2,V_3,V_5)		
V_6	∞	∞	∞	10 (V_1,V_2,V_3,V_4,V_6)	9 $(V_1,V_2,V_3,V_5,V_4,V_6)$
S	$\{V_2\}$	$\{V_2,V_3\}$	$\{V_2,V_3,V_5\}$	$\{V_2,V_3,V_5,V_4\}$	$\{V_2,V_3,V_5,V_4,V_6\}$

图 7.31　最短路径的求解过程

7.5.2　全源最短路径



视频讲解

问题：对于一个各边权值均大于零的赋权图，对每一对顶点 V_i，V_j 且 $V_i \neq V_j$，求出顶点 V_i 与顶点 V_j 之间的最短路径和最短路径长度。

为求每一对顶点间的最短路径，可以每次以一个顶点为源点，重复执行 Dijkstra 算法 n 次，其时间复杂度为 $O(n^3)$。此外，弗洛伊德提出弗洛伊德算法，其时间复杂度仍是 $O(n^3)$，但形式上较为简单。

弗洛伊德的核心思想是：以图的邻接矩阵为基础，通过不断地在图的每对顶点中插入顶点的方式更新邻接矩阵中任意两点间的最短路径，从而计算出图中任意顶点间的最短路径。除了用来计算最短路径距离的邻接矩阵以外，算法还需要一个同等大小的辅助矩阵记录最短路径的路线，该矩阵为路由矩阵。

假设求从顶点 V_i 到顶点 V_j 的最短路径。如果从 V_i 到 V_j 有弧，则从 V_i 到 V_j 存在一条长度为 edges[i][j]的路径，该路径不一定是最短路径，需要进行 n 次探测。首先考虑路径(V_i,V_0,V_j)是否存在(即判断(V_i,V_0)和(V_0,V_j)是否存在)。若存在该路径，则比较(V_i,V_j)和(V_i,V_0,V_j)的路径长度取长度较短者作为此刻 V_i 到 V_j 中间顶点序号不大于 0 的最短路径，即说明 V_i 到 V_j 的最短路径上经过 V_0 顶点。以此类推，在路径上再增加一个顶点 V_1，判断从 V_i 到 V_j 是否包含顶点 V_1 的路径，如果没有，则从 V_i 到 V_j 中间顶点序号不大于 1 的最短路径即为前面求出的从 V_i 到 V_j 中间顶点序号不大于 0 的最短路径。若从 V_i 到 V_j 的路径通过顶点 V_1，则将$(V_i,\cdots,V_1)$加上$(V_1,\cdots,V_j)$与从 V_i 到 V_j 中间顶点不大于 0 的最短路径进行比较，取其短者为当前 V_i 到 V_j 中间顶点序号不大于 1 的最短路径。以此类推，直到所有顶点全部加入，求得 V_i 到 V_j 的最短路径为止。

因此，为了计算最短路径问题，首先定义一个矩阵 D^{-1}(表示每对顶点间的直接路径即邻接矩阵)，$D^0,D^1,\cdots,D^n$。其中，D^k[i][j]表示从 V_i 到 V_j 中间顶点序号不大于 k 的最短路径长度。由于图中顶点序号不大于 n-1，所以 D^{n-1}[i][j]表示从 V_i 到 V_j 的最短路径长度。若从 V_i 到 V_j 没有中间顶点，即 D^{-1}[i][j]，则它恰好等于边长 edges[i][j]。于是，这个算法依次产生的矩阵序列为 $D^{-1},D^0,\cdots,D^{n-1}$。

假定已经计算出 D^{k-1}[i][j]，对于 D^k[i][j]，可根据如下两种情况进行计算。

- D^k[i][j]=G. edges[i][j]。
- D^k[i][j]=min{D^{k-1}[i][j], D^{k-1}[i][k]+ D^{k-1}[k][j]}($1\leqslant k\leqslant n$)。

弗洛伊德算法如下。

```
void Floyd(MGraph g)
{ int A[MAXV][MAXV],path[MAXV][MAXV] ;
 int i,j,k;
 for(i=0;i<g.n;i++)
 for(j=0;j<g.n;j++)
 {
  A[i][j]=g.edges[i][j];
  if(i!=j&&g.edges[i][j]<INF)
    path[i][j]=i;                    //顶点 i 到顶点 j 有边时
  else
    path[i][j]=-1;                   //顶点 i 到顶点 j 无边时
 }
 for(k=0;k<g.n;k++)
```

```
    { for(i=0;i<g.n;i++)
      for(j=0;j<g.n;j++)
      if(A[i][j]>A[i][k]+A[k][j])
      { A[i][j]=A[i][k]+A[k][j];
        path[i][j]=path[k][j];          //修改最短路径
      }
    }
  Dispath(g,A,path);                    //输出最短路径
}
```

以下函数用于输出所有顶点之间的最短路径。

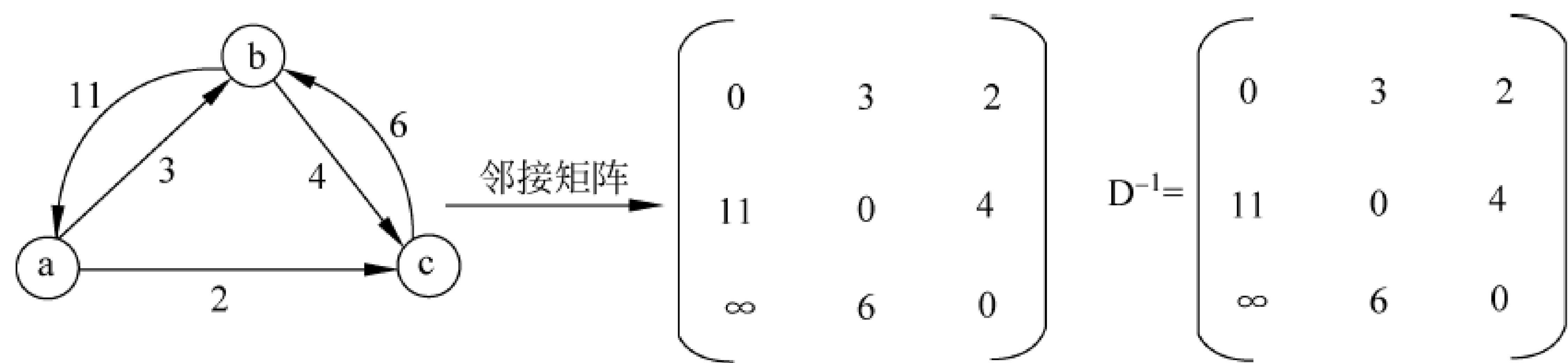
```
void Dispath(MGraph g,int A[][MAXV],int path[][MAXV])
{ int i,j,k,s;
  int apath[MAXV],d;                //存放一条最短路径的中间顶点(反向)及其顶点个数
  for(i=0;i<g.n;i++)
    for(j=0;j<g.n;j++)
    { if(A[i][j]!=INF&&i!=j) //若顶点 i 和顶点 j 之间存在路径
      { printf("输出从%d 到%d 的路径为:",i,j);
        k=path[i][j];
        d=0;apath[d]=j;             //路径上添加终点
        while(k!=-1&&k!=i)          //路径上添加中间点
      { d++;
          apath[d]=k;
          k=path[i][k];
      }
      d++;apath[d]=i;               //路径上添加起点
      printf("%d",apath[d]);        //输出起点
      for(s=d-1;s>=0;s--)           //输出路径上的中间顶点
          printf(",%d",apath[s]);
      printf("\t 路径长度为:%d\n",A[i][j]);
    }
  }
}
```

弗洛伊德算法的时间复杂度为 $O(n^3)$，其中 n 为图中的顶点个数。

例如，利用上述算法计算图 7.32 (a)所示的带权有向图的每一对顶点之间的最短路径，其路径长度如图 7.32(b)所示。

【例 7.12】 如图 7.33 所示，求赋权无向图中所有顶点间的最短路径。

因此，顶点 V_2 到顶点 V_6 的最短路径长度为 5，其最短路径为 V_2，V_3，V_6，其余顶点间的最短路径类似。



(a) 带权有向图G及邻接矩阵

$$D^0=\begin{pmatrix}0 & 3 & 2\\ 11 & 0 & 4\\ \infty & 6 & 0\end{pmatrix}\qquad D^1=\begin{pmatrix}0 & 3 & 2\\ 11 & 0 & 4\\ 17 & 6 & 0\end{pmatrix}\qquad D^2=\begin{pmatrix}0 & 3 & 2\\ 11 & 0 & 4\\ 17 & 6 & 0\end{pmatrix}$$

在每对顶点之间添加顶点a　　在每对顶点之间添加顶点b　　在每对顶点之间添加顶点c

$$P^{-1}=\begin{pmatrix}-1 & ab & ac\\ ba & -1 & bc\\ ca & cb & -1\end{pmatrix}\qquad P^0=\begin{pmatrix}-1 & ab & ac\\ ba & -1 & bc\\ ca & cb & -1\end{pmatrix}\qquad P^1=\begin{pmatrix}-1 & ab & ac\\ ba & -1 & bc\\ cba & cb & -1\end{pmatrix}$$

(b) 数组D和P取值的变化过程

图 7.32　弗洛伊德算法计算过程

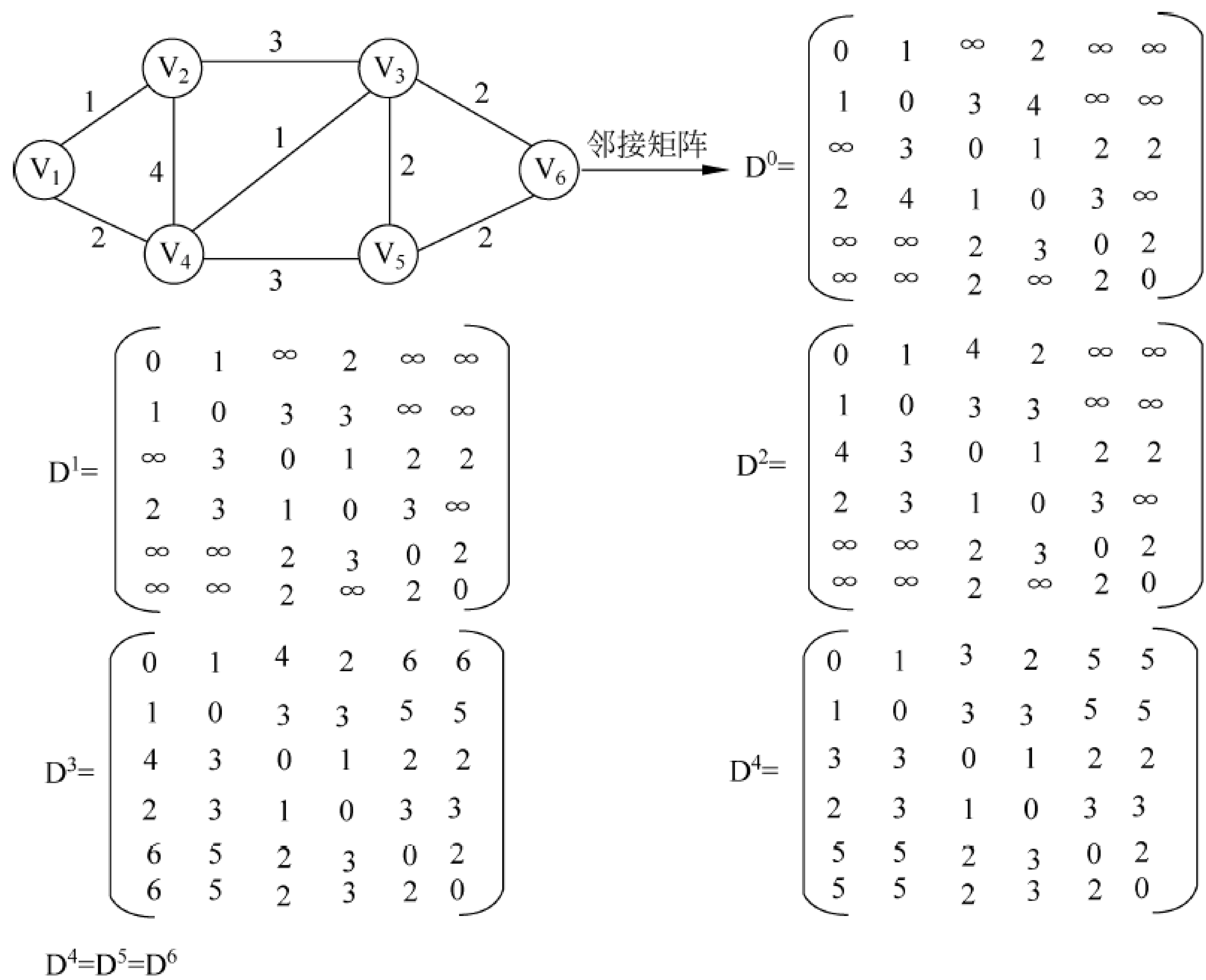


图 7.33　赋权无向图及所有顶点间的最短路径

7.6 活动网络

7.6.1 用顶点表示活动的AOV网络

设 G=(V,E)是一个具有 n 个顶点的有向图,V 中顶点序列 $V_1,V_2,\cdots,V_n$ 为一个拓扑序列,当且仅当该顶点序列满足下列条件:若 $<V_i,V_j>$ 是图中的边(即从顶点 V_i 到顶点 V_j 有一条路径),则在序列中顶点 V_i 必须排在顶点 V_j 前。在一个有向图中找一个拓扑序列的过程称为**拓扑排序**。

例如,计算机专业的学生必须完成一系列规定的基础课和专业课才能毕业,假设这些课程的名称与相应代号有如下关系,见表 7.2。

表 7.2 课程名称与相应代号的关系

课程代号	课程名称	先修课程
C_1	Java 基础编程	无
C_2	Java 高级编程	C_1
C_3	离散数学	C_1
C_4	数据结构	C_2,C_3
C_5	编译原理	C_2,C_4
C_6	操作系统	C_4,C_7
C_7	计算机组成原理	C_2

课程之间的先后关系有向图,如图 7.34 所示。

对这个有向图进行拓扑排序,可得到拓扑序列 C_1-C_3-C_2-C_4-C_7-C_6-C_5,也可得到拓扑序列 C_1-C_2-C_3-C_4-C_7-C_5-C_6,还可以得到其他的拓扑序列。学生可以按照任何一个拓扑序列的顺序进行课程学习。

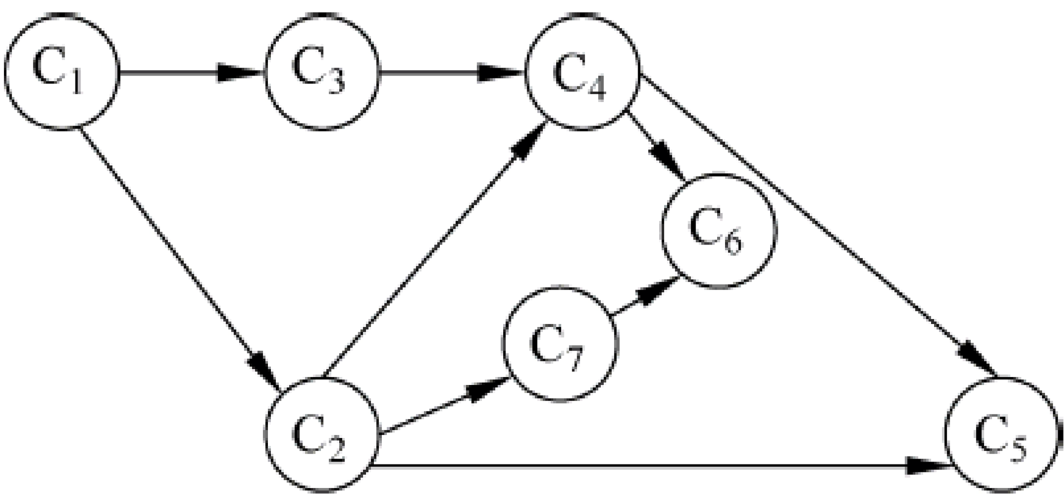


图 7.34 课程之间的先后关系有向图

拓扑排序方法如下。

step1:从有向图中选择一个没有前驱(即入度为 0)的顶点并且输出它。

step2:从图中删去该顶点,并且删去从该顶点出发的全部有向边。

step3:重复上述两步,直到所有顶点全部输出或碰到有向回路停止。

这样操作的结果有两种:一种是图中全部顶点全部输出,这说明图中不存在有向回路;另一种是图中顶点未被全部输出,剩余的顶点均有前驱顶点,这说明剩余图中存在有向回路。

为了实现拓扑排序的算法,对于给定的有向图,采用邻接表作为存储结构,为每个顶点设立一个链表,每个链表有一个表头结点,这些表头结点构成一个数组,表头结点中增加一个存放顶点入度的域 count,即将邻接表定义中的 VNode 类型修改如下。

```
typedef struct              //表头结点类型
{ Vertex data;              //顶点信息
  int count;                //存放顶点入度
  ArcNode * firstarc;       //指向第一条边
}VNode;
```

在执行拓扑排序的过程中，当某个顶点的入度为零(没有前驱顶点)时，就将此顶点输出，同时将该顶点的所有后继顶点(邻接点)的入度减 1。为了避免重复检测入度为零的顶点，可设立一个栈 St，存放入度为零的顶点。执行拓扑排序的算法如下。

```
void TopSort(ALGraph * G)
{ int i,j;
 int St[MAXV],top=-1;            //栈 St 的指针为 top
 ArcNode * p;
 for(i=0;i<G->n;i++)              //入度置初值 0
 G->adjlist[i].count=0;
 for(i=0;i<G->n;i++)              //求所有顶点的入度
 { p=G->adjlist[i].firstarc;
 while(p!=NULL)
 { G->adjlist[p->adjvex].count++;
   p=p->nextarc;
  }
 }
for(i=0;i<G->n;i++)
  if(G->adjlist[i].count==0)      //入度为 0 的顶点进栈
  { top++;
    st[top]=i;
  }
while(top>-1)                     //栈不为空时循环
{ i=St[top];top--;                //出栈
  printf("%d",i);                 //输出顶点
  p=G->adjlist[i].firstarc;       //找第一个相邻顶点
  while(p!=NULL)
  { j=p->adjlist[j].count;
  G->adjlist[j].count--;
  if(G->adjlist[j].count==0)      //入度为 0 的相邻顶点进栈
  { top++;
    St[top]=j;
  }
   p=p->nextarc;                  //找下一个相邻顶点
  }
 }
}
```

【例 7.13】 给出如图 7.35 所示的有向图 G 的全部可能的拓扑排序序列。

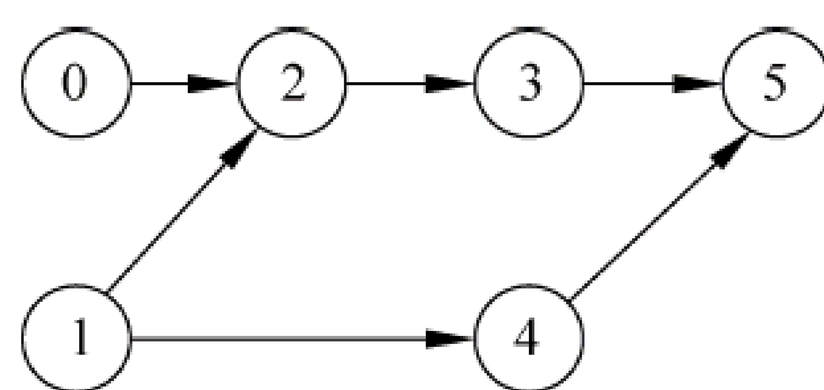


图 7.35 一个有向图 G

解：从图 G 中看到，有两个顶点的入度为 0，即顶点 0 和顶点 1，若先考虑顶点 0，删除顶点 0 及相关边，入度为 0 者有顶点 1；删除顶点 1 及相关边，入度为 0 者有顶点 2 和顶点 5；考虑顶点 2，删除顶点 2 及相关边，入度为 0 者有顶点 3 和顶点 4……如此得到拓扑序列：012345，012435，014235。

再考虑顶点 1，类似地，得到拓扑序列：102345，102435，104235，140235。

因此，所有的拓扑序列为：012345，012435，014235， 102345，102435，104235，140235。

7.6.2 AOE 图与关键路径



若用前面介绍过的**带权有向图**描述工程的预计进度，顶点表示**事件**，有向边表示**活动**，边 e 的权 dut(e)表示完成活动 e 所需的时间（如天数）或者活动 e 维持的时间。有向图中入度为 0 的顶点表示工程的**开始事件**（又称**源点**），出度为 0 的顶点表示工程的**结束事件**（又称**汇点**）。这样的有向图称为 **AOE**(Activity On Edge)**网**。

通常，每个工程都只有一个开始事件和一个结束事件，因此表示工程的 AOE 网都只有一个入度为 0 的顶点（称为**源点**）和一个出度为 0 的顶点（称为**汇点**）。如果图中存在多个入度为 0 的顶点，只要加一个虚拟源点，使这个虚拟源点到原来所有入度为 0 的点都有一条长度为 0 的边，将图变成只有一个源点即可。对存在多个出度为 0 的顶点的情况，可作类似处理。所以，只需讨论单源点和单汇点的情况。

利用这样的 AOE 图，能够计算完成整个工程预计需要多少时间，并找出影响工程进度的“关键路径”，从而为决策者提供修改各活动的预计进度依据。

在 AOE 网中，从源点到汇点的所有路径中，最具最大路径长度的路径称为**关键路径**。完成整个工程的最短时间就是网中关键路径的长度，也就是网中关键路径上各活动持续时间的总和，通常把关键路径上的活动称为**关键活动**。因此，只要找出 AOE 网中的关键活动，也就找到了关键路径。注意，在一个 AOE 网中，可以有不止一条关键路径。

例如，图 7.36 表示某工程的 AOE 网，共有 7 个事件和 8 个活动，其中 A 表示**源点**，G 表示**汇点**。

下面介绍如何利用 AOE 网计算完成整个工程最少需要的时间（即工程的工期时间），同时找出影响工程进度的关键活动。关键路径的长度是整个工程所需的最短工期。也就是说，要缩短整个工期，必须加快关键活动的进度。

利用 AOE 网进行工程管理时需要解决的主要问题如下。

- 计算完成整个工期的最短时间。
- 确定关键路径，找出哪些活动是影响工程进度的关键活动。

在 AOE 网中，若存在两条首尾相接的边 a_i＝<v,w>和 a_j＝<w,z>，则称活动 a_i 是活动 a_j 的前驱活动，活动 a_j 是活动 a_i 的后继活动。一个活动可能有多个前驱活动和多个后继活动。

显然，只有活动 a_j 的所有前驱活动都完成，事件 w 才发生（这里，w 是边 a_j 的头），即活动 a_j 才可以开始。如图 7.37 所示，当活动 a_h、活动 a_k 和活动 a_i 都完成时，事件 w 就发生了，活动 a_j 就可以开始了，事件 w 称为活动 a_j 的触发事件。

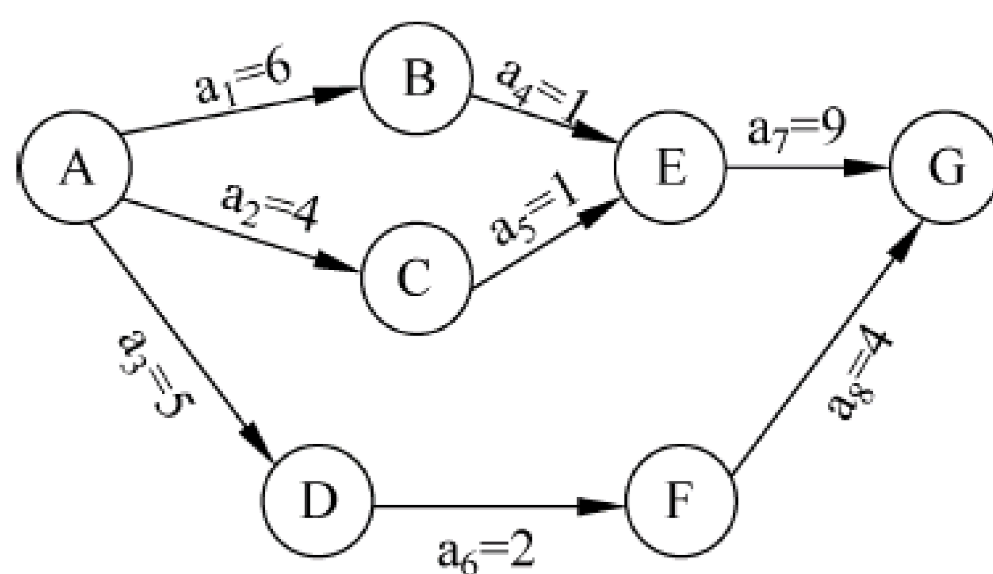


图 7.36 AOE 网的示例

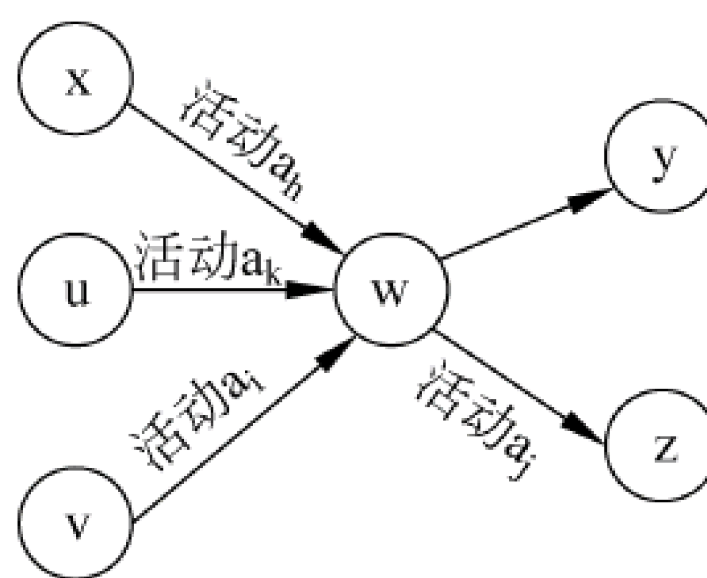


图 7.37 前驱活动和后继活动

为了在 AOE 网中找出关键路径，需要定义几个参量，并说明它们的计算方法。

1. 事件的最早发生时间 ve[k]

ve[k]是指从源点到顶点 k 的最长路径长度代表时间。这个时间决定了所有从顶点 k 出发的有向边代表的活动能够开工的最早时间。根据 AOE 网的性质，只要进入顶点 k 的所有活动＜j,k＞都结束时，顶点 k 代表的事件才能发生；而活动＜j,k＞的最早结束时间为 ve[j]＋dut(＜j,k＞)。所以，计算 k 顶点的最早发生时间方法如下。

$$\begin{cases} ve[\text{开始事件}] = 0 \\ ve[k] = \text{Max}\{ve[j] + dut(<j,k>)\} \quad <j,k> \in p[k] \end{cases}$$

其中，p[k]表示所有到达顶点 k 的有向边集合；dut(＜j,k＞)为有向边＜j,k＞上的权值。

2. 事件的最晚发生时间 vl[k]

vl[k]是指在不推迟整个工期的前提下，事件 k 允许的最晚发生时间。设有向边＜k, j＞代表从顶点 k 出发的活动，为了不拖延整个工程的工期，事件 k 发生的最迟时间必须保证不推迟从事件 k 出发的所有活动＜k, j＞的终点 j 的最迟时间 vl[j]。vl[k]的计算方法如下。

$$\begin{cases} vl[\text{结束时间}] = ve[\text{结束事件}] \\ vl[k] = \text{Min}\{vl[j] - dut(<k,j>)\} \quad <k,j> \in s[k] \end{cases}$$

其中，s[k]为所有从 k 顶点发出的有向边的集合。

3. 活动 a_i 的最早开始时间 e[i]

若活动 a_i 由弧＜k,j＞表示，根据 AOE 网的性质，只有事件 k 发生了，活动 a_i 才能开始。也就是说，活动 a_i 的最早开始时间等于事件 k 的最早发生时间，因此有

$$e[i] = ve[k]$$

4. 活动 a_i 的最晚开始时间 l[i]

活动 a_i 的最晚开始时间是指在不推迟整个工程完成日期的前提下必须开始的最晚时间，若由弧＜k,j＞表示，则 a_i 的最晚开始时间要保证事件 j 的最迟发生时间不拖后腿，因此应该有

$$l[i] = vl[j] - dut(<k,j>)$$

根据每个活动的最早开始时间 e[i]和最晚开始时间 l[i]，就可以判定该活动是否为关键活动，也就是那些 l[i]＝e[i]的活动就是关键活动，而那些 l[i]＞e[i]的活动则不是关键路径上的关键活动，l[i]－e[i]的值为活动的时间余额。关键活动被确定之后，关键活动所在的路径就是关键路径。

5. 活动 a_i 的最迟开始时间 l[i]－最早开始时间 e[i]＝d[i]该活动完成的时间余量

它是在不增加完成整个工程所需的总时间的情况下，活动 a_i 可以拖延的时间。若活动的时间余量等于 0，说明该活动为关键活动；若活动的时间余量大于 0，说明该活动为非关键活动。

例如，在图 7.36 中，A 到 E 的最长路径是 A-B-E，其长度等于 6＋1＝7，所以事件 E 的最早发生时间等于 7。图 7.36 的一条关键路径是 A-B-E-G，其长度等于 6＋1＋9＝16，于是，完成整个工程至少需要 16 天(假定时间单位是天)。

为使事件 v_i 尽早发生，从源点 v_1 到事件 v_i 的最长路径上的活动必须“刻不容缓”地发

生,一旦触发事件,便立即开始以该顶点为起点的活动,而且应当在规定时间内完成,否则后序事件就不能按时发生,影响整个工程进度。例如,图 7.36 中的活动 a_4,一旦事件 B 发生,活动 a_4 必须立即开始。

对那些并不处在最长路径上的活动来说,即使稍稍推迟一些时间完成,也对工程的进度无碍。例如,图 7.36 中,活动 a_5 不处在事件 E 的最长路径上,只要它在第七天之前完成,就不影响事件 E 的发生,由于路径 A-C-E 的长度等于 4+1=5,所以活动 a_5(或者说活动 a_5 连同活动 a_2),可有 7−5=2 天的富余时间。

【例 7.14】 求图 7.36 所示的 AOE 网的关键路径。

解:对于图 7.36 所示的 AOE 网,源点为顶点 A,汇点为顶点 G。计算各事件 e 的 ve(e)如下。

```
ve[A] = 0
ve[B] = ve[A] + dut [a1]= 6
ve[C] = ve[A] + dut [a2] = 4
ve[D] = ve[A] + dut [a3]= 5
ve[E] = MAX{ve[B] + dut [a4], ve[C] + dut [a5]} = MAX{7,5} =7
ve[F] = ve[D] + dut [a6] = 7
ve[G] = MAX{ve[E] + dut [a7], ve[F] + dut [a8]} = MAX{16,11} = 16
```

计算各事件 e 的 vl[e]如下。

```
vl[G] = ve[G] = 16
vl[F] = vl[G] − dut [a8] = 12
vl[E] = vl[G] − dut [a7] = 7
vl[D] = vl[F] − dut [a6] = 10
vl[C] = vl[E] − dut [a5] = 6
v1[B] = vl[E] − dut [a4] = 6
v1[A] = MIN{v1[B] − dut [a1],vl[c] − dut [a2],vl[D] − dut [a3]} = MIN{0,2,5} = 0
```

计算各活动 a_i 的 e[a_i]、l[a_i]和 d[a_i]如下。

```
活动 a1: e[a1] = ve[A] = 0    l[a1] = vl[B] − 6 = 0    d[a1] = 0;
活动 a2: e[a2] = ve[A] = 0    l[a2] = vl[C] − 4 = 2    d[a2] = 2;
活动 a3: e[a3] = ve[A] = 0    l[a3] = vl[D] − 5 = 5    d[a3] = 5
活动 a4: e[a4] = ve[B] = 6    l[a4] = v1[E] − 1 = 6    d[a4] = 0;
活动 a5: e[a5] = ve[C] = 4    l[a5] = vl[E] − 1 = 6    d[a5] = 2;
活动 a6: e[a6] = ve[D] = 5    l[a6] = vl[F] − 2 = 10   d[a6] = 5;
活动 a7: e[a7] = ve[E] = 7    l[a7] = vl[G] − 9 = 7    d[a7] = 0
活动 a8: e[a8] = ve[F] = 7    l [a8] = vl[G] − 4 = 12  d[a8] = 5;
```

由此可知,关键活动有 $a_1 \to a_4 \to a_7$,因此关键路径有一条:A—B—E—G,如图 7.38 所示。

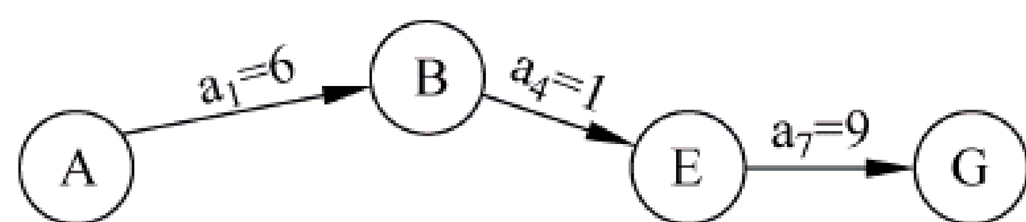


图 7.38 关键路径

由上述方法得到计算关键路径的算法步骤如下。

step1:输入 e 条弧<j,k>,建立 AOE 网络图并存储。

step2:从源点 v1 出发,令 ve[1]=0,按照拓扑有序的求其余各个顶点的最早发生时间 ve[i](2≤i≤n),如果得到的拓扑有序序列中顶点个数小于网中顶点数 n,则说明网中存在

环，不能求关键路径，算法提前终止；否则执行步骤 step3。

step3：从终点 vn 出发，令 vl[n]＝ve[n]，逆拓扑有序求其余各个顶点的最迟发生时间 vl[i]（n－1⩾i⩾1）。

step4：根据各个顶点的 ve[ ]最早发生时间和 vl[ ]最晚发生时间值求每条弧 s 的最早开始时间 e[s]和最迟开始时间 l[s]。若某条弧满足条件 e[s]＝l[s]，则为关键活动。

如上所述，为了计算关键路径，可以一边进行拓扑排序，一边计算各个顶点的 ve[i]，因此，可以如前面给出的拓扑排序算法中那样设置一个存放入度为 0 顶点的链式栈，利用图邻接表中的 count 数组作为链式栈的存储空间，实现拓扑排序。在入度为 0 的顶点出栈同时进行反向拉链，以便在计算完各个顶点的 ve[i]之后，可以按照拓扑有序的顺序计算各顶点的 vl[i]，但在程序中，为了简化算法，假定在求关键路径之前已经对各个顶点实现拓扑排序，并按拓扑有序的顺序对各顶点重新进行了编号，算法在求 ve[i]，i＝1，2，…，n 时，按拓扑有序的顺序计算，在求 vl[i]，i＝n，n－1，…，1 时，按逆拓扑有序的顺序计算。最后扫描一遍邻接表，计算 e[ ]和 l[ ]。

7.7 综合案例

7.7.1 道路修建问题

1. 问题描述

现有 N 个村子，这些村子的编号为 1～N。要修建一些路，这些路应把这些村子两两连接起来。如果 A 村与 B 村相连，那么要么村 A 与村 B 之间有一条路，要么 A 村与 C 村之间有一条路，并且 B 村与 C 村之间也有一条路。

已知村子与村子之间已经修建了一些路，这里的任务是修建剩下的路，以使所有的村子都连接上，并且路的总长度最短。

【输入】 第 1 行包含一个整数 N(3⩽N⩽100)，N 是村子的数量。接下来有 N 行，这 N 行中的第 i 行包含 N 个整数，这 N 个整数中的第 j 个代表了村与村之间的距离（这个距离应在区间[1，1000]内），即村 i 与村之间的距离。接下来是一个整数 Q(0⩽Q⩽N×(N＋1)/2)，然后是 Q 行数据，每一行包含两个整数 a 与 b(1⩽a＜b⩽N)，这意味着村 a 与村 b 之间的路已修建。下面给出一段输入示例。

```
3
0 990 692
990 0 179
692 179 0
1
1 2
```

【输出】 程序应在一行中输出一个整数,这个数代表了要把这些村子连接起来所需的最短总路程。例如,若按照上述给出的示例进行输入,将得到输出179。

2. 解题思路

显然,这是一个在图中求最小生成树的问题,按照Kruskal算法在图中求解最小生成树即可成功解决该问题。

3. 代码实现

```
#include <stdio.h>
#include <iostream>
using namespace std;
#define N 110

typedef struct {
  int prior, next, dis;
} Node;
Node node[N * N];
int father[N];
int find(int);
int union_(int , int);
void MST_Kruskal();
int n, ans = 0;
int cmp(const void * a, const void * b) {
  Node * x = (Node * )(a);
  Node * y = (Node * )(b);
  return (x->dis - y->dis);
}
int main(int argc, char * * argv) {
  int i, j, a, b, t = 0, m = 0;
  scanf("%d", &n);
  for(i = 1; i < n + 1; i++) {
    node[t].prior = i;
    node[t].next = j;
    scanf("%d", &node[t++].dis);
  }
  scanf("%d", &m);
  for(i = 0; i < N; i++)
    father[i] = -1;
  for(i = 0; i < m; i++) {
    scanf("%d%d", &a, &b);
    union_(a, b);
  }
  qsort(node, n * n, sizeof(Node), cmp);
  MST_Kruskal();
  cout << ans << endl;
  return 0;
}
void MST_Kruskal() {
```

```
    int i = 0, u, v;
    for(i = 0; i < n * n; i++) {
        u = node[i].prior;
        v = node[i].next;
        if(find(u) != find(v)) {
            union_(u, v);
            aus += node[i].dis;
        }
    }
}
int find(int a) {                    //并查集搜索父结点
    int i, temp;
    for(i = a; father[i] > 0; i = father[i]);
    while(a != i) {
        temp = father[a];
        father[a] = i;
        a = temp;
    }
    return i;
}
int union_(int a, int b) {
    int temp, temp1 = find(a), temp2 = find(b);
    if(temp1 == temp2) return 0;
    temp = father[temp1] + father[temp2];
    father[temp2] = temp1;
    father[temp2] = temp;
    return 1;
}
```

7.7.2 回家路线问题

1. 问题描述

贝茜在外面的田里，她现在想回到谷仓，因为明早约翰会叫醒她并要她一起去挤牛奶，所以她希望回到谷仓后能睡得长久些。因此她想尽快从外面的田里赶回来。

约翰的田里有 N(2≤N≤1000)个地标，并且每个地标都标了唯一的号码，这些号码是 1～N 的数字。1 号表示谷仓，贝茜现在所在的苹果林的标识号码是 N。已知牛在田地里的 T(1≤T≤2000)个道路耕作，这些道路是位于两个地标之间的双向通路。贝茜对自己认路的能力不是很自信，所以一旦她从某个地标出发，开始沿道路走时，她总是会一直走到道路结束的地方（也就是下一个地标处）。请编写一个程序，帮助贝茜确定回到谷仓的最短距离。这里假设回到谷仓的路径必然存在。

【输入】 本程序的输入由两部分组成：第一部分即程序输入的第 1 行，是两个整数 T 和 N；第二部分由第 2 行到第 T+1 行组成，每行是 3 个由空格隔开的整数，前两个是地标的号码，第 3 个是该条道路的长度，其范围是 1～100。下面给出一段输入示例。

```
5 5
1 2 20
2 3 30
3 4 20
4 5 20
1 5 100
```

【输出】 本程序的输出是一个整数，它表示贝茜从N号地标到1号地标必须经过的最短路径，以上面的输入为例，此时需要的最短路径是90。可以知道，贝茜所走的最短路径为5→4→3→2→1。

2. 解题思路

本题是一道非常典型的求最短路径的题目，这里使用Dijkstra算法解决问题。

3. 代码实现

```
#include <stdio.h>
#include <memory.h>
#include <iostream>
using namespace std;

const int MAXN = 1001;
const int INFI = 2147483647;
int map[MAXN][MAXN];
int dij[MAXN], n;
void read() {
  int t, i, j, x, y, temp;
  scanf("%d%d", &t, &n);
  for (i = 0; i < MAXN; i++) {
    dij[i] = INFI;
    for (j = 0; j < MAXN; j++)
      map[i][j] = INFI;
    map[i][i] = 0;
  }
  for (i = 0; i < t; i++) {
    scanf("%d%d%d", &x, &y, &temp);
    if (map[x][y] > temp) map[y][x] = map[x][y] = temp;
  }
}

void dijkstra() {
  bool used[MAXN];
  int i, j, k, min;
```

```
    memset(used, 0, sizeof(used));
    used[1] = true;
    for (i = 0; i <= n; i++)
      dij[i] = map[1][i];
      for (i = 1; i <= n; i++) {
        min = INFI;
        for (j = 1; j <= n; j++)
          if (!used[j] && min > dij[j]) min = dij[j], k = j;
        used[k] = true;
        for (j = 1; j <= n; j++)
          if(!used[j] && map[k][j] != INFI &&
          dij[j] > min + map[k][j]) dij[j] min + map[k][j];
    }
    cout << dij[n] << endl;
}
int main() {
    read();
    dijkstra();
    return 0;
}
```

7.7.3 棍子还原问题

1. 问题描述

乔治拿来一组等长的棍子，将它们随机地裁断(截断后的小段称为木棒)，使得每一节木棒的长度都不超过 50 个长度单位。然后他又想把这些木棒恢复为裁断前的状态，但忘记了棍子的初始长度。请设计一个程序，帮助乔治计算棍子的可能最小长度。每一节木棒的长度都用大于零的整数表示。

输入：由多个案例组成，每个案例包括两行：第一行是一个不超过 64 的整数，表示裁断之后共有多少节木棒；第二行是裁断后得到的各节木棒的长度。在最后一个案例之后，是零。

输出：为每个案例分别输出木棒的可能最小长度，每个案例占一行。

输入样例：

```
9
5 2 1 5 2 1 5 2 1
4
1 2 3 4
0
```

输出样例：

```
6
5
```

2. 解题思路

本题是一道典型的深度优先搜索题目。为避免搜索空间过大,耗费太长时间,可设计如下剪枝策略。

step1:如果用了若干节木棒后,剩余的长度还等于假设的长度,就说明没办法扩展出合法的拼接。

step2:已知第 i 根木棍不可构成成功的拼法,若剩下长度等于第 i 根木棍的长度,则说明无法继续深度优先搜索。

step3:如果某次拼接选择长度为 L1 的木棒,导致最终失败,则在同一位置尝试下一根木棒时,要跳过所有长度为 L1 的木棒。

step4:拼每一根棍子的时候,确保已经拼好的部分的长度从长到短排列。

3. 实现代码

```
#include <stdio.h>
#include <string.h>
#include <algorithm>
using namespace std;
int stick[100];
int len, n;
bool use[100];
//比较函数:用于将木棍从大到小排序
bool cmp(int a, int b) {
  return a > b;
}

//unused:没有使用的棍子的数目
//left:剩下的长度
//preno:保证木棍在拼的过程中,是从长到短扩展的
bool dfs(int unused, int left, int preno/*剪枝4*/) {
  //所有的棍子已经用了,且没有剩余的长度,符合搜索条件
  if(unused == 0 && left == 0) return true;
  //若没有剩下的,则新开一条棍子
  if(left == 0) {
    left = len;
    preno = 0;
  }
  if(preno != 0)
    preno += 1;
  //寻找没有使用过的棍子
  for(int i = preno; i < n; i++) {
    if(i > 0 && use[i-1] == false && stick[i] == stick[i - 1])
      continue;          //剪枝3
    //找到没有用过的,而且长度比 left 值要小(能够填进去)
    if(!use[i] && stick[i] <= left) {
      use[i] = true;
      //若在当前情况下能够搜索出正确答案,则返回 true
```

```
            if(dfs(unused - 1, left - stick[i], i))
                return true;
            //否则不使用当前的棍子
            use[i] = false;
            if(left == stick[i] || len == left)
                return false;                //剪枝2和剪枝1
        }
    }
    return false;
}

int main() {
    int i, sum;
    while (scanf("%d", &n) && n) {
        memset(len, 0, sizeof(len));
        for(i = sum = 0; i < n; i++) {
            scanf("%d", &stick[i]);
            sum += stick[i];
        }
        sort(len, len + n, cmp);
            //根据题目条件,从最长的一节棍子开始搜索,直至长度总和
            for (i = stick[0]; i <= sum; i++) {
                //棍子总长被i整除才进行搜索,否则跳过
                if(sum % i != 0) continue;
                memset(use, false, sizeof(use));
                len = i;
                if (dfs(n, i, 0)) {
                printf("%d\n", i);
                break;
            }
        }
    }
    return 0;
}
```

本章小结

本章主要介绍了图结构基本知识,主要学习要点如下。

- 掌握图的定义及相关概念,包括图、有向图、无向图、完全图、子图、连通图、度、入度、出度、简单回路和环等的定义。
- 理解图的各种存储结构,包括邻接矩阵和邻接表、十字链表等。
- 掌握图的基本运算,包括创建图、输出图、深度优先遍历、广度优先遍历等。
- 掌握图的其他运算,包括最小生成树、最短路径、拓扑排序和关键路径等的算法。
- 灵活运用图解决一些综合应用问题。

第5篇　数据运算篇

第 8 章　查　　找

经常利用计算机完成的一项工作是查找信息。查找也是许多计算机应用程序的核心操作,因此,研究对存储的数据如何高效率地进行查找操作是非常必要的,在一些实时查询系统中尤其如此。排序也是计算机中经常进行的一种操作,其目的是将一组无序的记录序列调整为按关键字有序的记录序列。生活中我们往往将排序和查找相结合,数据对象被排列成有序的序列后,可便于人们进行查找操作。

查找又称为检索,是指在某种数据结构中找出满足给定条件的元素。查找是一种十分有用的操作。例如,在学生成绩表中查找某个学生的成绩元素,在图书馆的书目文件中查找某编号的图书元素等。本章介绍各种常用的查找算法。

8.1　查找的基本概念

被查找的对象是由一组元素组成的表或文件,而每个元素由若干个数据项组成,假设每个元素都有一个能唯一标识该元素的**关键字**,在这种条件下,**查找**的定义是:给定一个值 k,在含有 n 个元素的查找表中找出关键字等于 k 的元素。若找到,则**查找成功**,返回该元素的信息或该元素在表中的位置;否则**查找失败**,返回相关的提示信息。

因为查找是对已存入计算机中的数据进行的运算,所以采用哪种查找方法,首先取决于使用哪种数据结构表示“查找表”,即表中元素是按何种方式组织的。为了提高查找速度,常常用某些特殊的数据结构组织表,或对表事先进行诸如排序这样的运算。因此,在研究各种查找方法时,首先必须弄清这些方法针对的数据结构(尤其是存储结构)是什么,对表中关键字的次序有何要求。例如,是对无序集合查找,还是对有序集合查找?

若在查找的同时对表做修改运算(如插入和删除),则相应的表称为**动态查找表**,反之称为**静态查找表**。查找也有内查找和外查找之分。若整个查找过程都在内存进行,则称之为**内查找**;反之,若查找过程中需要访问外存,则称之为**外查找**。

由于查找的主要运算是关键字的比较,所以通常把查找过程中对关键字的**平均比较次数**(也称为平均查找长度)作为衡量一个查找算法效率优劣的标准。平均查找长度(Average Search Length,ASL)定义为

$$ASL = \sum_{i=1}^{n} p_i c_i$$

其中,n 是所在查找表中元素的个数;p_i 是查找第 i 个元素的概率(一般地,认为每个元素的查找概率相同,即 $p_i=\frac{1}{n}(1\leqslant i\leqslant n)$);$c_i$ 是找到第 i 个元素时所需的比较次数。显然,c_i 取

决于算法，p_i 取决于实际应用。

大多数情况下，查找成功的可能性比不成功的可能性大得多，特别是查找表中元素的个数 n 很大时，查找不成功的概率可以忽略不计。如果查找不成功的情况不能忽略，平均查找长度应该是查找成功时平均查找长度加上查找不成功时的平均查找长度。假设查找成功与不成功的概率相等，都是 1/2，则平均查找长度的计算公式为

$$ASL = \sum_{i=1}^{n} p_i c_i + \sum q_j d_j$$

其中，q_j 是查找关键字等于某个给定值失败的概率；d_j 是查找失败时的比较次数。由于很难估计一共有多少个数据元素查找失败，所以一般估算出查找失败时的平均比较次数 $ASL_{失败}$，再假设查找成功时为等概率查找，则有

$$ASL = \frac{1}{2n}\sum_{i=1}^{n} c_i + \frac{1}{2} ASL_{失败}$$

另外，衡量查找算法还要考虑算法所需要的存储空间和算法的时间复杂度等问题，但**默认**是计算查找方法的 $ASL_{成功}$。

8.2 静态表的查找

在表的组织方式中，静态表是最简单的一种。本节将介绍 4 种在静态表上进行查找的方法，它们分别是顺序查找、有序表的折半查找、有序表的斐波那契查找、分块查找，四者在查找的同时均**不对**表做修改。查找与数据的存储结构有关，静态表$\{a_1, a_2, \cdots, a_n\}$有顺序和链式两种存储结构。为了突出查找方法本身，本节只介绍以顺序表作为存储结构时实现的顺序查找算法。

被查找的顺序表类型定义如下。

```
#define MAXL 100
typedef int KeyType;                     //假设关键字为整数
typedef char InfoType[10];
typedef struct
{ KeyType key;                           //KeyType 为查找关键字的数据类型
  InfoType data;                         //其他数据
} NodeType;
typedef NodeType SeqList[MAXL];          //顺序表类型
```

8.2.1 顺序查找

顺序查找又称线性查找，是一种最简单的查找方法。它的基本思路是：从表的一端开始，顺序扫描静态表，依次将扫描到的关键字和给定值 k 相比较，若当前扫描到的关键字与 k 相等，则查找成功；若扫描结束，仍未找到关键字等于 k 的元素，则查找失败。

顺序查找的算法如下(在顺序表 R[0..n−1]中查找关键字为 k 的元素，第 n 号元素为哨兵，成功时返回找到的元素的逻辑序号，失败时返回 0)。

```
int SeqSearch(SeqList R,int n,KeyType k) {
R[n].key = k;                    //设置哨兵
int i = 0;
while (R[i].key != k)            //从表尾往前找
  i++;
if (i >= n)
  return -1;
else
  return i;
}
```

从顺序查找过程可以看到，c_i（查找元素 a_i 所需的关键字比较次数）取决于元素 a_i 在表中的位置。如查找表中第 1 个元素 R[0]时，仅需比较一次，即 $c_i=1$；查找表中第 n 个元素 R[n−1]时，需比较 n 次，即 $c_i=n$。因此，查找成功时的顺序查找的平均查找长度为

$$ASL_{成功}=\sum_{i=1}^{n}p_ic_i=\frac{1}{n}\sum_{i=1}^{n}i=\frac{1}{n}\times\frac{n(n+1)}{2}=\frac{n+1}{2}$$

即查找成功时的平均比较次数约为表长的一半。算法的时间复杂度为 O(n)。

若 k 值不在表中，则需进行 n 次比较之后，才能确定查找失败，所以查找不成功时的平均查找长度为 n，即 $ASL_{失败}=n$。

顺序查找的优点是算法简单，且对表的结构无任何要求，无论是用顺序表，还是用链表存放元素，也无论元素之间是否按关键字有序，它都同样适用。顺序查找的缺点是查找效率低，因此，当 n 较大时，不宜采用顺序查找。

注意：若查找表中设置了"**监视哨**"，则顺序查找关键字 k 值失败时，比较次数为 n+1 次，即 $ASL_{失败}=n+1$。

8.2.2　折半查找


视频讲解

折半查找又称二分查找，它是一种效率较高的查找方法。但是，折半查找有两个前提条件：查找表是有序表，即表中元素按关键字有序（在下面的讨论中，假设是递增有序的），并且是基于顺序存储结构进行待查找元素的存储。这两个条件缺一不可。例如，元素值有序的单链表不能进行折半查找，因为它是链式存储。

折半查找的基本思路是：设 R[low..high]是当前的查找区间，首先确定该区间的中点位置 $mid=\lfloor(low+high)/2\rfloor$，然后将待查的 k 值与 R[mid].key 比较，比较结果分为以下 3 种。

- 若 R[mid].key=k，则查找成功，并返回该元素在查找表的逻辑序号。
- 若 R[mid].key>k，则由表的有序性可知 R[mid..n−1].key 均大于 k，因此若表中存在关键字等于 k 的元素，则该元素必定在位置 mid 左边的子表 R[0..mid−1]中，故新的查找区间是左子表 R[0..mid−1]。
- 若 R[mid].key<k，则要查找的 k 必定在 mid 的右子表 R[mid+1..n−1]中，即新的查找区间是右子表 R[mid+1..n−1]。

下一次查找是针对新的查找区间进行相似的查找。

因此，可以从初始的查找区间 R[0..n−1]开始，每经过一次与当前查找区间的中点位置上的关键字的比较，就可确定查找是否成功，若不成功，则查找区间缩小一半。重复这一

过程,直至找到关键字为 k 的元素,或者直至当前的查找区间为空(即查找失败)时为止。

其算法如下(在有序表 R[0..n-1]中进行折半查找,成功时返回元素在查找表的逻辑序号,失败时返回-1)。

```
int BinSearch(SeqList R, int n, KeyType k)
{
int low = 0, high = n - 1, mid;
while (low <= high)
{
  mid = low + (high - low) / 2;
  if (R[mid].key == k)             //查找成功返回
    return mid;
  if (R[mid].key > k)              //继续在 R[low..mid-1]中查找
  high = mid - 1;
  else
  low = mid + 1;                   //继续在 R[mid+1..high]中查找
  }
return -1;
}
```

折半查找过程可用二叉树描述,把当前查找区间的中点位置上的元素作为根,左子表和右子表中的元素分别作为根的左子树和右子树,由此得到的二叉树称为描述折半查找的**判定树**或**比较树**。

注意:判定树的形态只与表元素个数 n 相关,而与输入实例中 R[0..n-1].key 的取值无关。

例如,具有 13 个元素(R[0..12])的有序表可用图 8.1 所示的二叉判定树表示。图中,圆形元素表示内部结点,内部结点中的数字表示该元素在有序表中的位置。方形结点表示外部结点,外部结点中的两个值表示查找不成功时关键字等于给定值的元素对应的元素序号的开区间,即外部结点中 i~j 表示被查找值 k 是介于 R[i].key 和 R[j].key 之间的,即 R[i].key<k<R[j].key。

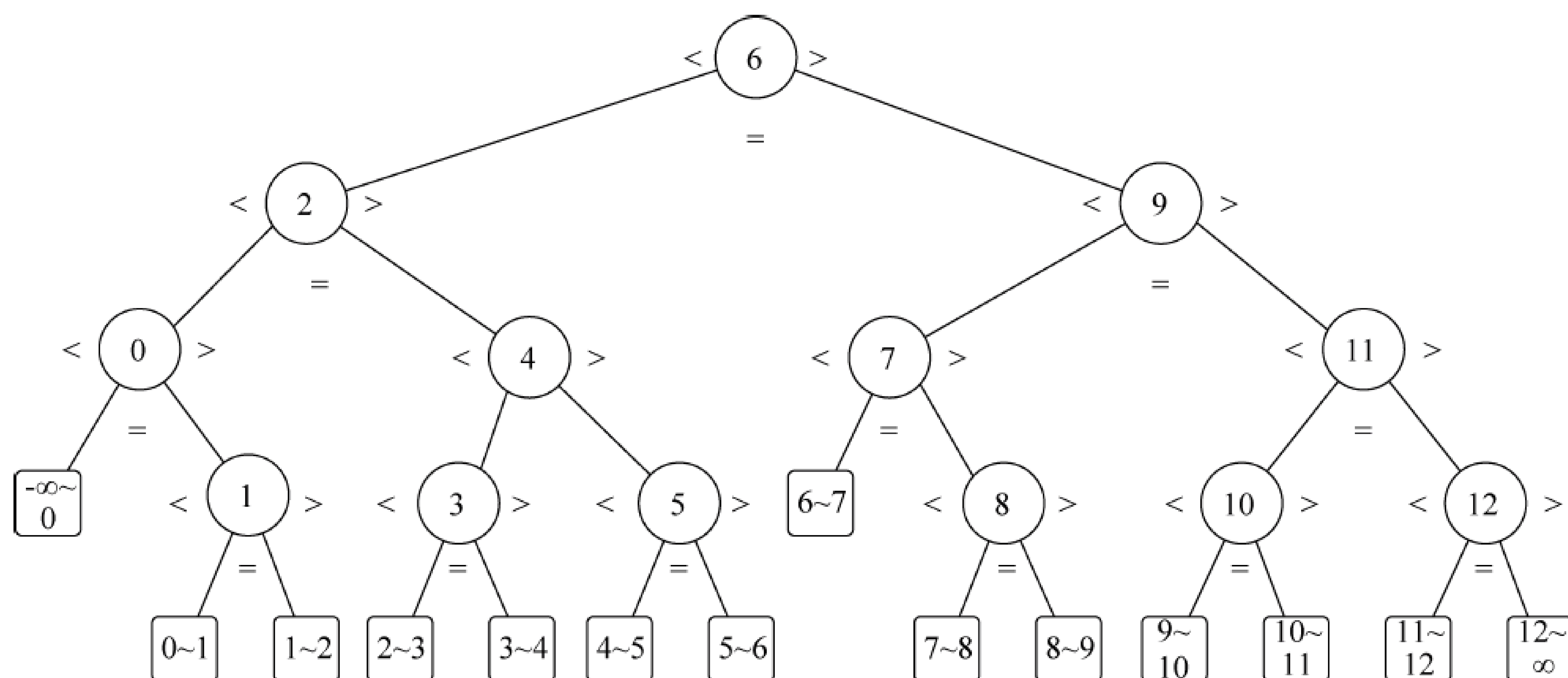


图 8.1　二叉判定树

显然，若查找的元素是表中第 7 个元素(R[6])，则只需比较一次；若查找的元素是表中第 3 个元素(R[2])或第 10 个元素(R[9])，则需比较两次；查找第 1，5，8，12 个元素需要比较 3 次；查找第 2，4，6，9，13 个元素需要比较 4 次。

由此可见，一次成功的折半查找过程恰为一条从判定树的根到被查元素的路径，而关键字的比较次数恰为所查找元素在树中的层数。若查找失败，则比较过程是一条从判定树的根到某个外部结点的路径，所需的关键字比较次数是该路径上内部结点的个数。

借助二叉判定树，很容易求得折半查找的平均查找长度。为讨论方便起见，不妨设内部结点的总数 $n=2^h-1$，即判定树是高度为 $h=\lceil \log_2(n+1)\rceil$ 的满二叉树(深度 h 不计入外部结点)。树中第 i 层上的元素个数为 $2^{i-1}(i\geqslant 1)$，查找该层上的每个元素需要进行 i 次比较。因此，在等概率假设下，折半查找成功时的平均查找长度为

$$ASL_{折半}=\sum_{i=1}^{n}p_i c_i=\frac{1}{n}\sum_{i=1}^{h}2^{i-1}\times i=\frac{n+1}{n}\times\log_2(n+1)-1\approx\log_2(n+1)-1$$

折半查找，在查找失败时所需比较的关键字个数不超过判定树的深度，在最坏情况下查找成功时的比较次数也不超过判定树的深度。因为判定树中度数小于 2 的元素只可能在最下面的两层上(不计外部结点)，所以 n 个元素的判定树的深度和 n 个元素的完全二叉树的深度相同，即为$\lceil \log_2(n+1)\rceil$。由此可见，折半查找的最坏性能和平均性能相当接近。虽然折半查找的效率高，但是须预先将表按关键字排序，而排序本身是一种很费时的运算，即使采用高效率的排序法，也要花费 $O(n\log_2 n)$的时间(参见第 9 章)。

另外，折半查找须确定查找的区间，因此只适用于顺序存储结构，不适用于链式存储结构。为保持表的有序，在顺序结构里插入和删除都必须移动大量的元素，因此，折半查找特别适用于那种一经建立就很少改动，而又经常需要进行查找的静态表。

【例 8.1】 给定 13 个数据元素的有序表{1，9，18，20，21，37，44，56，70，76，83，86，97}，若采用折半查找，试问：

(1) 若查找给定值为 20 的元素，将依次与表中哪些元素比较？

(2) 若查找给定值为 26 的元素，将依次与表中哪些元素比较？

(3) 假设查找表中每个元素的概率相同，求查找成功时平均查找长度和查找不成功时的平均查找长度？

解：折半查找判定树如图 8.2 所示。(其中，圆形结点左、右、下方的数值分别对应折半查找算法中 low、high、mid 的实时取值，方形结点下的两数值表示定位关键元失败的情况，即当 low <= high 条件不成立时，low、high 相应的取值。这些值在折半插入排序算法中派得上用场，届时将再作回顾。)

(1) 若查找给定值为 20 的元素，依次与表中元素 **44，18，21，10** 比较，共比较 4 次，成功。

(2) 若查找给定值为 59 的元素，依次与元素 **44，76，56** 比较，共比较 3 次，失败。

(3) 查找成功时，会找到图中某个圆形结点，则成功时的平均查找长度：

$$ASL_{成功}=\frac{1\times1+2\times2+4\times3+6\times4}{13}\approx3.154$$

(4) 查找不成功时，会找到图中某个方形结点，则不成功时的平均查找长度：

$$ASL_{失败}=\frac{2\times3+12\times4}{14}=3.857$$

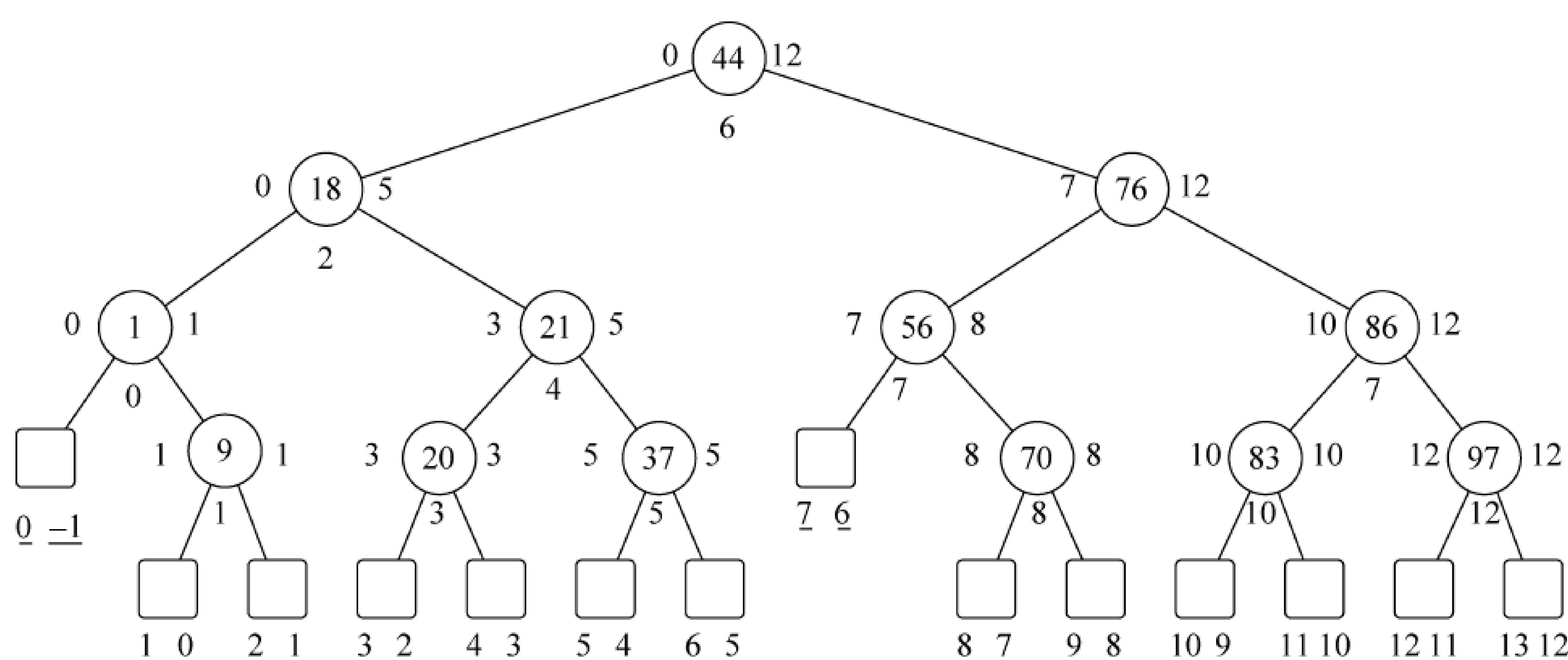


图 8.2 折半查找判定树

8.2.3 斐波那契查找

斐波那契查找也是对有序表进行查找。与折半查找选择中间元素的方法不同，斐波那契查找是根据斐波那契序列对表进行分割。

斐波那契序列为 0,1,1,2,3,5,8,13,21,…，即有 $F_0=0$，$F_1=1$，$F_i=F_{i-1}+F_{i-2}(i\geqslant 2)$。通过上述的递推公式，可以得到斐波那契序列中任意两个相邻的斐波那契数的差的绝对值还是斐波那契数。假设开始时表中元素的个数 n 比某个斐波那契值小 1(如果 n 不是恰好满足条件，则可以增加一些虚元素)，满足表达式 $n=F_u-1$，在区间 $[F_j+1,\ F_u-1](0<j<u)$ 上将给定值 k 与 $r[F_{u-1}]$.key 进行比较，有 3 种情况。

- 若 $k<r[F_{u-1}]$.key，则在 $[F_j+1,\ F_u-1]$ 查找。
- 若 $k>r[F_{u-1}]$.key，则在区间 $[F_{u-1}+1,\ F_u-1]$(此表的长度变为 $F_{u-2}-1$)查找。
- 若关键字与 $r[F_{u-1}]$.key 相等，则表示查找成功。

例如，有一个长度为 12 的有序序列，即 n=12，则有 $12=F_7-1$，所以首先使关键字序列 k 与序列中的第 F_6 个元素(即第 8 个元素)比较，若 $k<r[F_6]$.key，则再使 k 与序列中的第 $F_7-F_6=F_5$ 个元素(即第 5 个元素)比较，若 $k>r[F_6]$.key，则再使 k 与序列中的第 $F_6+(F_6-F_5)$ 个元素(即第 11 个元素)比较，以此类推。上述的查找过程也可以用一个二叉树表示，这个二叉树称为斐波那契树，如图 8.3 所示。在这个斐波那契树中，总共有 F_u-1 个结点，根结点为 F_{u-1}，根的左子树有 $F_{u-1}-1$ 个结点，右子树有 $F_{u-2}-1$ 个结点。

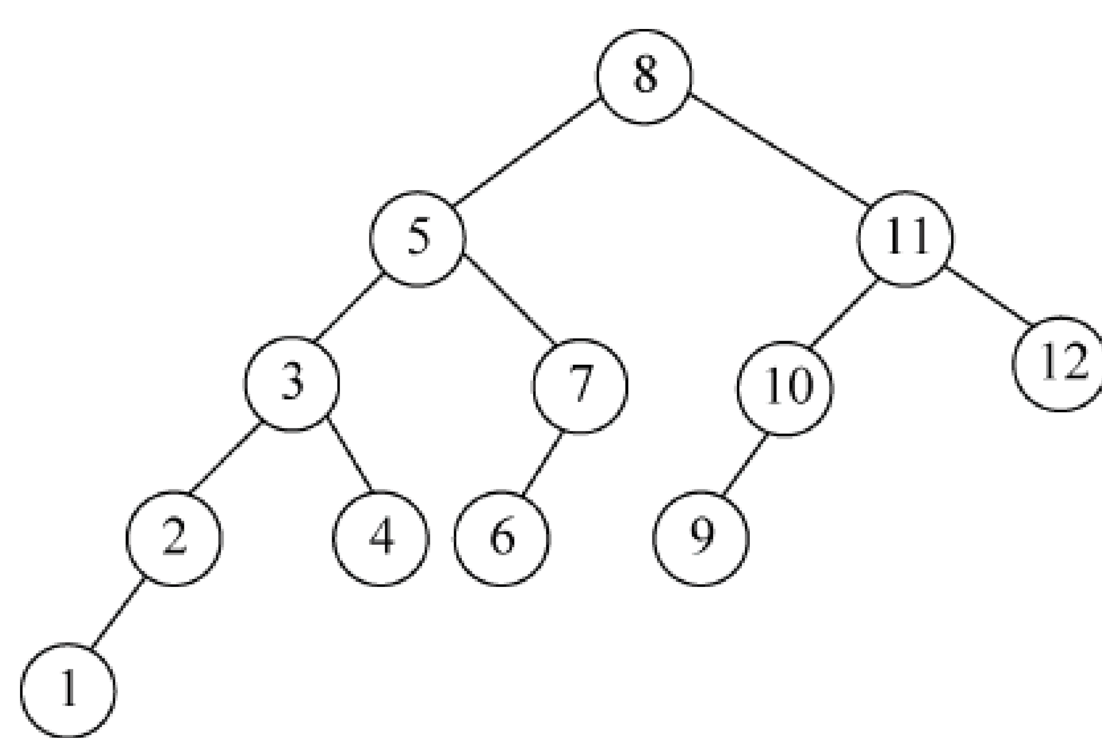


图 8.3 12 个结点的斐波那契树

斐波那契查找的平均性能比折半查找好，但最坏情况下的性能比折半查找差，其时间复杂度仍为 $O(\log_2 n)$。此外，斐波那契查找在计算查找位置时只进行加、减运算，而加、减运算要优于折半查找的乘、除运算。

8.2.4 分块查找

分块查找又称**索引顺序查找**,它是顺序查找方法的改进,其目的是通过缩小查找范围改进顺序查找的性能。这种方法是将顺序查找分为若干个子表块 $b_1, b_2, \cdots, b_n$,并要求当 $i<j$ 时,b_i 子表中的记录关键字都小于 b_j 中的记录关键字,但 b_i 子表内部元素的值可以无序,即被称为“块内无序,块间有序”的分割方式。

分块后,辅助创建一个索引表,每个分割块在索引表中有一项,称为索引项。索引项由两部分组成:一个是块内记录关键字的最大值;另一个是块的第一个记录关键字在索引表中的位置。索引项在索引表中按关键字有序方式组织。分块查找的索引结构如图 8.4 所示。

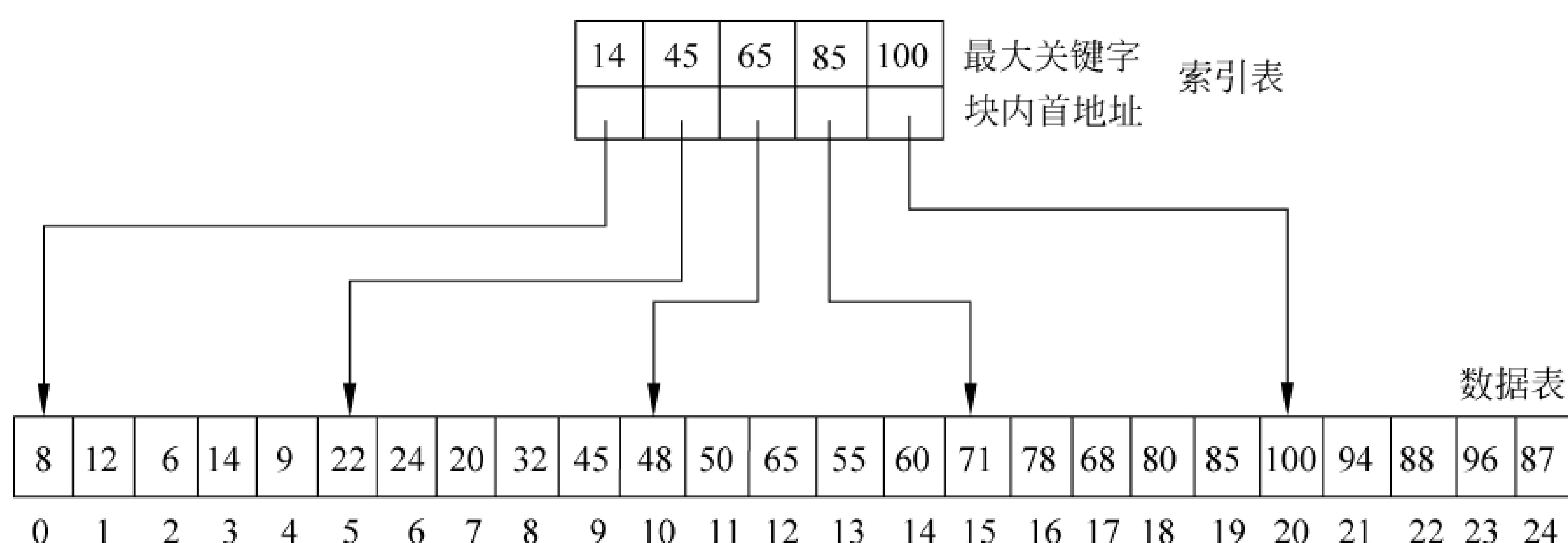


图 8.4 分块查找的索引结构

假设有一个索引表,其中包含 25 个元素,其关键字序列为(8,12,6,14,9, 22,24,20,32,45,48,50,65,55,60,71,78,68,80,85,100,94,88,96,87),假设将这 25 个元素分成 5 块(b=5),每块内有 5 个元素(s=5),该线性表的索引存储结构如图 8.4 所示。第一块中最大的元素值是 14,小于第二块中的最小元素值 20,第二块中最大的元素值是 45,小于第三块中的最小元素值 48,第三块中最大的元素值是 65,小于第四块中的最小元素值 68,第四块中最大的元素值是 85,小于第五块中的最小元素值 87。

分块索引查找过程分为如下两步。

step1:将待查找的关键字 k 和索引表中的关键字进行比较,以确定待查记录所在的块。索引表中的查找可以用顺序查找,也可以用折半查找(索引表中关键字的值是递增有序的)。

step2:进一步用顺序查找方法,在相应的块内进行逐一关键字的比较,确认关键字 k 是否存在于数据表中。

例如,在上述的索引顺序表中查找 32。首先,将 32 与索引表中的关键字进行比较,若采用顺序查找,因为 $14<32<45$,所以若查找 32,无须到第一块中查找,应该到第二块内查找,在第二块中逐一进行元素的比较,最后查找到第二块中的第 4 个元素等于 32,因而得出查找成功的结论,且比较次数为 $2+4=6$ 次。

分块查找的平均查找长度由两部分组成,即查找索引表的平均查找长度 ASL_b,以及在相应的块内进行顺序查找的平均查找长度 ASL_s。

$$ASL_{分块} = ASL_b + ASL_s$$

假定将长度为 n 的查找表分成 b 块,且每块含 s 个元素,则 b=⌈n/s⌉。又假定表中每个元素的查找概率相等,则每个索引项的查找概率为 1/b,块中每个元素的查找概率为 1/s。若用顺序查找法确定待查找元素所在的块,则有

$$\begin{aligned}ASL_{分块} &= ASL_b + ASL_s \\ &= \frac{1}{b}\sum_{j=1}^{b} j + \frac{1}{s}\sum_{i=1}^{s} i = \frac{b+1}{2} + \frac{s+1}{2} = \frac{1}{2}\left(\frac{n}{s}+s\right)+1\end{aligned}$$

可见,此时的 ASL 不仅和表长 n 有关,也和每一块中的记录的个数 s 有关,而且在给定 n 的前提下,s 是可以选择的。数学方法容易证明,当 $s=\sqrt{n}$ 时,$ASL_{分块}$ 可取最小值 $\sqrt{n}+1$(此时 $b=s=\sqrt{n}$)。由此发现,理想状态下的分块查找比顺序查找有了很大的改进,但不及折半查找。

注意:若对索引表采用折半查找,则有 $ASL_{分块}=\log_2\left(\frac{n}{s}+1\right)+\frac{s}{2}$。

8.3 动态查找表

从 8.2 节的讨论可知,当用静态表作为表的组织形式时,可以有 4 种查找法,其中折半查找效率最高。但由于折半查找要求表中元素按关键字有序,且不能用链表作存储结构,因此,当表的插入或删除操作频繁时,为维护表的有序性,需要移动表中的大量元素。这种由移动元素引起的额外时间开销,就会降低折半查找的效率,即折半查找**只适用**于静态查找表。

若要对动态查找表进行高效率的查找,可采用本节介绍的几种特殊的二叉树或树作为表的组织形式,这里将它们统称为树表。下面分别讨论在这些树表上进行查找和修改操作的方法。

8.3.1 二叉排序树


视频讲解

二叉排序树(Binary Sort Tree,BST)又称**二叉查找(搜索)树**,其定义为:二叉排序树或者是空树,或者是满足如下性质的二叉树。

- 若它的左子树非空,则左子树上所有元素的值均小于根元素的值。
- 若它的右子树非空,则右子树上所有元素的值均大于根元素的值。
- 左、右子树本身又各是一棵二叉排序树。

上述性质简称二叉排序树性质(BST 性质)。由 BST 性质可知,对于二叉排序树中的任一元素 R,其左(右)子树中任一元素 s(若存在)的关键字必小(大)于 R 的关键字。需要指出的是,如此定义的二叉排序树中,各元素关键字是**唯一**的。

但实际应用中,不能保证被查找的数据集中各元素的关键字互不相同,所以可将二叉排序树定义中 BST 性质(1) 里的「小于」改为『小于等于』,或将 BST 性质(2) 里的「大于」改为『大于等于』,甚至可同时修改这两个性质。

从 BST 性质可推出二叉排序树的另一个**重要性质**:中序遍历二叉排序树所得到的中序序列是一个**递增有序序列**。

在讨论二叉排序树上的运算前，定义其结点的类型如下。

```
typedef int KeyType;
typedef char InfoType[10];
typedef struct node {                    //记录类型
KeyType key;                             //关键字项
InfoType data;                           //其他数据域
struct node *lchild, *rchild;            //左、右孩子指针
} BSTNode;
```

1. 二叉排序树的插入和生成

在二叉排序树中插入一新元素，要保证插入后仍满足 BST 性质。算法 InsertBST()的插入过程如下。

step1：若二叉排序树 T 为空，则创建一个 key 值为 k 的结点，将它作为根结点。

step2：否则将 k 和根结点的关键字进行比较，

若两者相等，则说明树中已有此关键字 k，无须插入，直接返回值 0；

若 k<T->key，则将 k 插到根结点的左子树中（递归调用自身），

否则将它插到右子树中（递归调用自身）。

```
int InsertBST(BSTNode *&p, KeyType k)
{
if (p == NULL)                           //原树为空，新插入的记录为根结点
{
  p = (BSTNode *)malloc(sizeof(BSTNode));
  p->key = k;
  p->lchild = p->rchild = NULL;
  return 1;
}
else if (k == p->key)
  return 0;                              //树中存在相同关键字的结点，返回 0
else if (k < p->key)
  return InsertBST(p->lchild, k);        //插入到 *p 的左子树中
else
  return InsertBST(p->rchild, k);        //插入到 *p 的右子树中
}
```

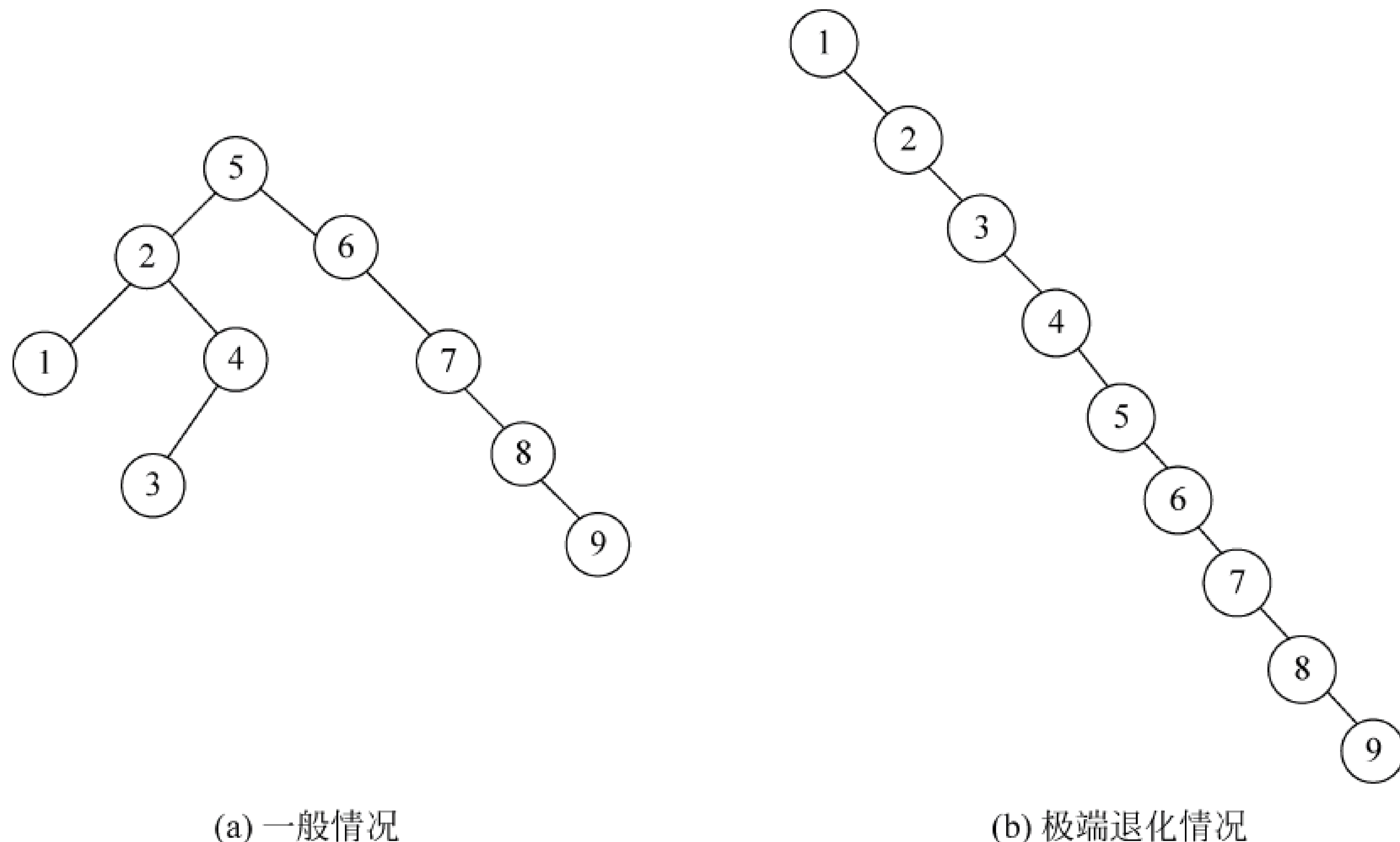
注意：上述算法是在根结点指针为 p(p 可能为空)的二叉排序树中插入一个关键字值为 k 的结点，p 的值可能发生变化，所以一定要用引用类型，即将 p 的值改变后的结果回传给实参，否则会出现错误。

二叉排序树的生成是从一个空树开始，每插入一个关键字，就调用一次插入算法，将它插入到当前已生成的二叉排序树中。从关键字数组 A[0..n-1]生成二叉排序树的算法 CreateBST()如下。

```
BSTNode * CreateBST(KeyType A[ ],int n)          //返回 BST 树根结点指针
{
BSTNode * bt = NULL;                             //初始时 bt 为空树
int i = 0;
while (i< n)
{
 InsertBST(bt, A[i]);                            //将关键字 A[i]插入二叉排序树 T 中
 i++;
}
return bt;                                       //返回建立的二叉排序树的根指针
}
```

每个结点插入时都需要从根结点开始比较,若比根结点的 key 值小,当前指针移到左子树,否则当前指针移到右子树,重复此操作,直到当前指针为空,再创建一个新结点,将当前指针指向它,这样便将这个结点插入到二叉排序树中了。因此,将任何结点插入到二叉排序树中,都是作为叶子结点插入的。

对于一组关键字集合,若关键字初始序列不同,上述算法生成的二叉排序树可能不同。例如,关键字序列为{5,2,1,6,7,4,8,3,9},上述算法生成的二叉排序树如图 8.5(a)所示;关键字序列为{1,2,3,4,5,6,7,8,9},上述算法生成的二叉排序如图 8.5(b)所示。显然,图 8.5(a)的二叉排序树的查找效率比图 8.4(b)的查找效率高。不难推知,构造的二叉排序树高度越小,其查找效率越高。8.4 节将讨论如何构造这种高查找效率的二叉排序树。



(a) 一般情况　　　　(b) 极端退化情况

图 8.5　不同序列产生的二叉排序树

因为二叉排序树的中序序列是一个有序序列,所以,对于任意一个关键字序列构造一棵二叉排序树,其实质是对此关键字序列进行排序,使其变为有序序列,「排序树」的名称也由此而来。通常将这种排序称为树排序,可以证明这种排序的平均时间复杂度为 $O(n\log_2 n)$。

2. 二叉排序树上的查找

因为二叉排序树可看作是一个有序表,所以在二叉排序树上进行查找,和折半查找类

似，也是一个逐步缩小查找范围的过程。递归查找算法 SearchBST( )如下(在二叉排序树 bt 上查找关键字为 k 的元素，成功时返回找到的元素结点指针，否则返回 NULL)。

```
BSTNode *SearchBST(BSTNode *bt,KeyType k)
{
if (bt==NULL || bt->key==k) return bt;              //递归终结条件
if (k<bt->key)
  return SearchBST(bt->lchild,k);                    //在左子树中递归查找
else
  return SearchBST(bt->rchild,k);                    //在右子树中递归查找
}
如果不仅要找到关键字为 k 的结点，还要找到其双亲结点，采用的递归查找算法如下.
/* 在 bt 中查找关键字为 k 的结点，若查找成功，则该函数返回该结点的指针，
 * f 返回其双亲结点；否则，该函数返回 NULL.
 * 其调用方法如下.
 * SearchBST1(bt, x, NULL, f);
 * 这里的第 3 个参数 f1 仅作中间参数，用于求 f，初始设为 NULL
 */
BSTNode *SearchBST1(BSTNode *bt, KeyType k, BSTNode *f1, BSTNode *&f)
{
if (bt == NULL)
{
  f = NULL;
  return(NULL);
}
else if (k==bt->key)
{
  f=f1;
  return(bt);
}
else if (k<bt->key)
  return SearchBST1(bt->lchild, k, bt, f); //在左子树中递归查找
else
  return SearchBST1(bt->rchild, k, bt, f); //在右子树中递归查找
}
```

显然，在二叉排序树上进行查找，若查找成功，则是从根结点出发走了一条从根结点到查找结点的路径；若查找不成功，则是从根结点出发走了一条从根到某个叶子结点的路径。因此，与折半查找类似，和关键字比较的次数不超过树的深度。

然而，折半查找法查找长度为 n 的有序表，其判定树是**唯一**的，而含有 n 个元素的二叉排序树却**不唯一**。对于含有同样一组元素的表，由于元素插入的先后次序不同，所构成的二叉排序树的形态和深度也可能**不同**。如图 8.5(a)、(b)所示的两棵二叉排序树的深度分别是 5 和 9，因此，在查找失败的情况下，在这两棵树上进行的关键字比较次数最多分别为 5 和 9；在查找成功的情况下，它们的平均查找长度也不相同。

对于图 8.5(a)所示的二叉排序树，在等概率假设下，查找成功的平均查找长度为

$$ASL_{成功} = \frac{1+2\times2+3\times3+4\times2+5\times1}{9} \approx 3$$

类似地,在等概率假设下,图 8.5(b)所示的二叉排序树在查找成功时的平均查找长度为

$$ASL_{成功} = \frac{1+2+3+4+5+6+7+8+9+9}{9} = 5$$

由此可见,在二叉排序树上进行查找时的平均查找长度和**二叉排序树的形态**有关。

在**最坏**情况下,二叉排序树是通过把一个有序表的 n 个元素依次插入而生成的,此时所得的二叉排序树**退化**为一棵深度为 n 的单支树,它的平均查找长度和在单链表上的顺序查找相同,即$(n+1)/2$。

在**最好**情况下,二叉排序树在生成的过程中,树的形态比较匀称,最终得到的是一棵形态与折半查找的判定树相似的二叉排序树,此时它的平均查找长度大约为$\log_2(n)$。

就平均时间性能而言,二叉排序树上的查找和折半查找差不多,但就维护表的有序性而言,前者更有效,因为无须移动元素,只修改指针即可完成对二叉排序树的插入和删除操作,且其平均执行时间均为 $O(\log_2 n)$。

【例 8.2】 已知一组待查关键字为{35, 19, 71, 12, 63, 39, 96, 94, 25, 58},按组中列出的元素顺序依次插入到一棵初始为空的二叉排序树中,画出该二叉排序树,并求在等概率情况下查找成功时的平均查找长度。

解:生成的二叉排序树如图 8.6 所示。

$$ASL_{成功} = \frac{1\times1+2\times2+4\times3+3\times4}{10} = 2.9$$

注意:在等概率情况下查找失败的平均查找长度为

$$ASL_{失败} = \frac{5\times3+6\times4}{11} \approx 3.5$$

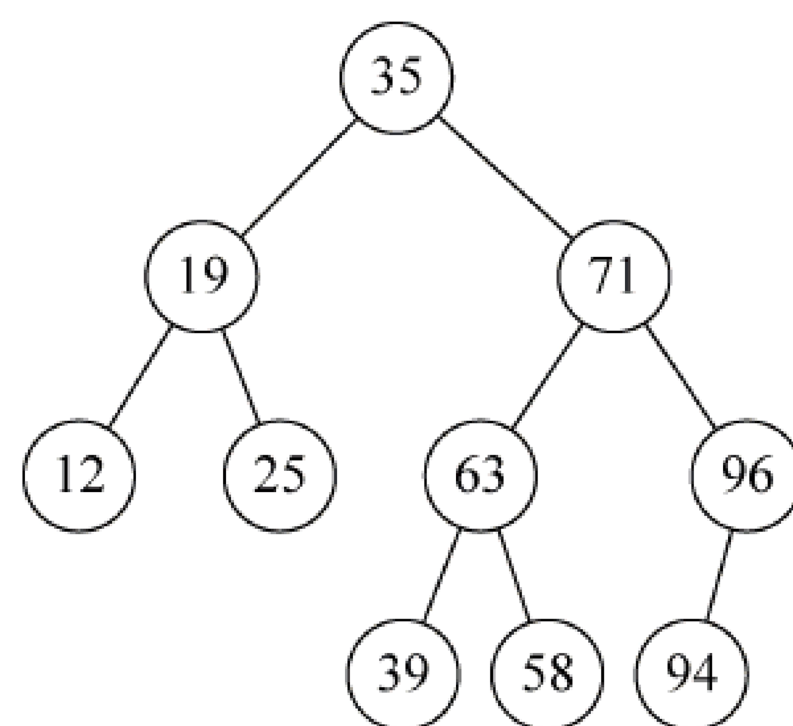


图 8.6 生成的二叉排序树

【例 8.3】 设计一个算法,对于给定的二叉排序树中的结点 *p,找出其左子树中的最大结点和右子树中的最小结点。

解:根据二叉排序树的定义,一棵二叉排序树中左子树的最大结点为根结点的最右下结点,右子树的最小结点为根结点的最左下结点。对应的算法如下。

```
void maxminnode(BSTNode *p)
{
if (p != NULL)
{
  if (p->lchild != NULL)
    printf("左子树的最大结点为:%d\n", maxnode(p->lchild));
  if (p->rchild != NULL)
    printf("右子树的最小结点为:%d\n", minnode(p->rchild));
  }
}

KeyType maxnode(BSTNode *p)
{                                          //返回一棵二叉排序树中的最大结点关键字
```

```
while (p->rchild != NULL)
  p = p->rchild;
return p->data;
}
KeyType minnode(BSTNode * p)
{                                         //返回一棵二叉排序树中的最小结点关键字
while (p->lchild != NULL)
  p = p->lchild;
return p->data;
}
```

3. 二叉排序树的删除

从二叉排序树中删除一个结点时,不能把以该结点为根的子树都删去,只能删除该结点本身,并且还要保证删除后所得的二叉树仍然满足 BST 性质。换言之,在二叉排序树中删去一个结点,就相当于删去有序序列(即该树表的中序序列)中的一个元素。

删除操作首先必须进行查找,假设在查找过程结束时已经保存了待删除结点及其双亲结点的地址。指针变量 del 指向待删除的结点,指针变量 par 指向待删除结点 del 的双亲结点。删除过程如下。

- 若待删除的结点是叶子结点,直接删去该结点。如图 8.7(a)所示,直接删除结点 8。这是最简单的删除结点的情况。
- 若待删除的结点只有左子树,而无右子树。根据二叉排序树的特点,可以直接将其左子树的根结点放在被删结点的位置。如图 8.7(b)所示,* del 为 * par 的右子树根结点,要删除 * del 结点,只需将 * del 的左子树(其根结点值为 10)作为 * par 的右子树。
- 若待删除的结点只有右子树,而无左子树。与上面情况类似,可以直接将其右子树的根结点放在被删结点的位置。如图 8.7(c)所示,* del 为 * par 的右子树根结点,要删除 * del 结点,只需将 * del 的右子树(其根结点值为 5)作为 * par 结点的右子树。
- 若待删除的结点同时有左子树和右子树。根据二叉排序树(中序遍历)的特点,可从其左子树中选择关键字最大的结点或从其右子树中选择关键字最小的结点,放在被删结点的位置。假设选取左子树上关键字最大的结点,则该结点一定是左子树的最右下结点。

注意:当把左子树中最右下结点 * rb 上移时,如果它有左子树,(必定无右子树),那么还需将这棵左子树改为 * rb 结点原来双亲结点的右子树。具体过程如图 8.7(d)所示,若要删除 * del(其关键字为 6)结点,找到其左子树最右下结点(其值为 5),用它代替 * del 结点,并将其原来的左子树(其根结点值为 4)作为其原来的双亲结点(其值为 2)的右子树。

删除二叉排序树结点的算法 DeleteBST()如下(指针变量 del 指向待删除的结点,指针变量 par 指向待删除结点 * del 的双亲结点)。

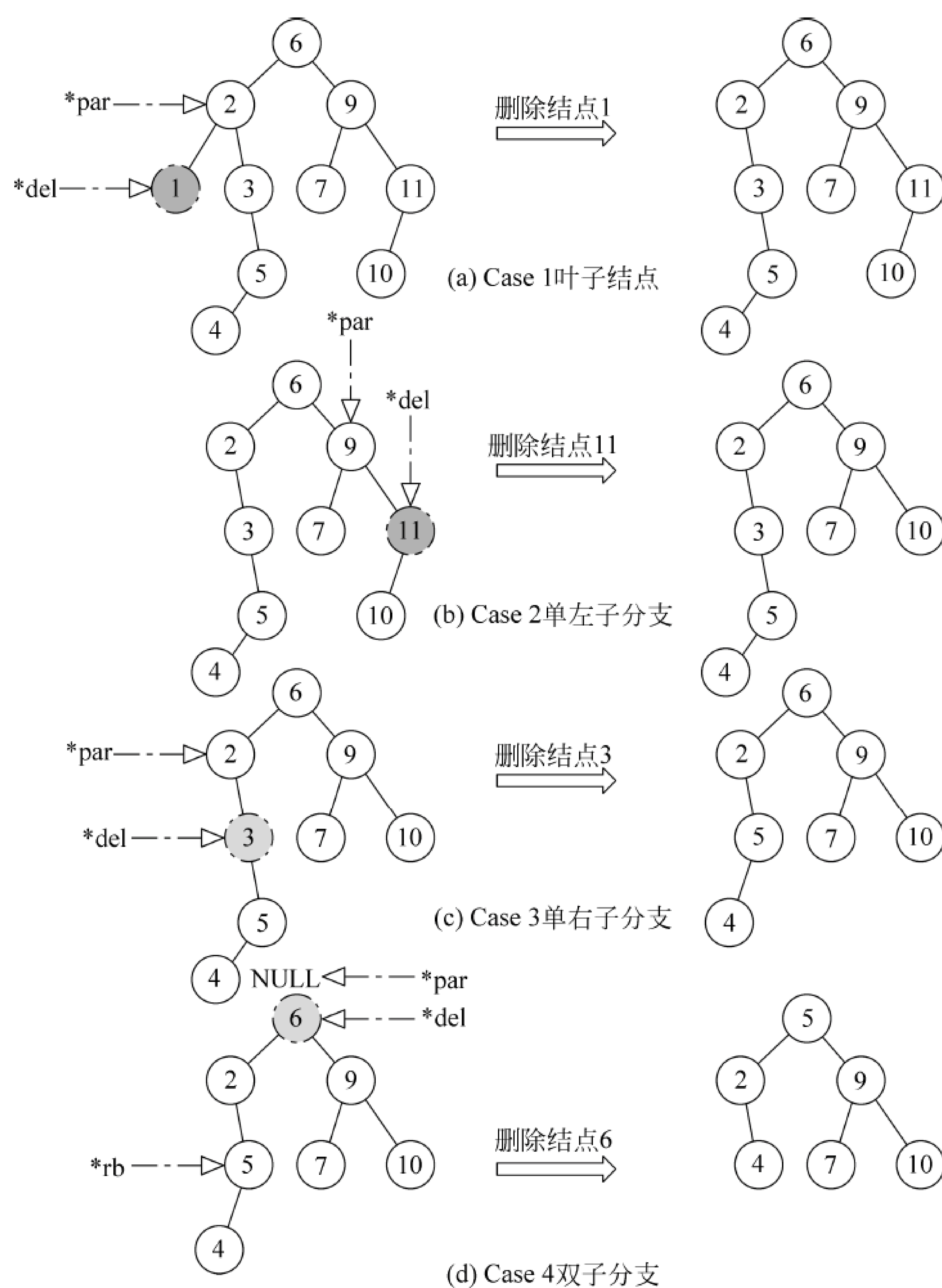


图 8.7　二叉排序树结点的删除

```
int DeleteBST(BSTNode * &bt, KeyType k)
{                                       //在 bt 中删除关键字为 k 的结点
if (bt == NULL) return 0;               //空树删除失败
else {
  if (k < bt→key)
      return DeleteBST(bt→lchild, k);   //递归在左子树中删除为 k 的结点
  else if (k > bt→key)
      return DeleteBST(bt→rchild, k);   //递归在右子树中删除为 k 的结点
  else {
      Delete(bt);                       //调用 Delete(bt)函数删除 * bt 结点
  return 1;
  }
 }
}
```

注意：上述算法是在根结点指针为 bt 的二叉排序树中删除一个结点，bt 的值可能发生变化，所以一定要用引用类型，才可将 bt 的值改变后的结果回传给实参，否则会出现错误。

```
void Delete(BSTNode * &del)
{                                  //从二叉排序树中删除 * del 结点
BSTNode * tmp;
if (del->rchild == NULL)
{                                  // * del 结点没有右子树的情况
  tmp = del;
  del = del->lchild;               //直接将其右子树的根结点放在被删结点的位置上
  free(tmp);
}
else if (del->lchild == NULL)
{                                  // * del 结点没有左子树的情况
  tmp = del;
  del = del->rchild;               //将 * del 结点的右子树作为双亲结点的相应子树
  free(tmp);
}
else
  Delete2(del, del->lchild);       // * del 结点既有左子树,又有右子树的情况
}

void Delete2(BSTNode * del, BSTNode * &rb)
{                                  //被删 * del 结点有左、右子树的删除过程
BSTNode * tmp;
if (rb->rchild != NULL)
  Delete2(del, rb->rchild);        //递归找最右下结点 * rb
else
{                                  //找到了最右下结点 * rb
  del->key = rb->key;              //将 * rb 的关键字值赋给 * del
  tmp = rb;
  rb = rb->lchild;                 //直接将其左子树的根结点放在被删结点的位置上
  free(tmp);                       //释放原 * rb 的空间
  }
}
```

例如，对于图 8.8 所示的二叉排序树，调用 DeleteBST(bt,17)的过程如下。

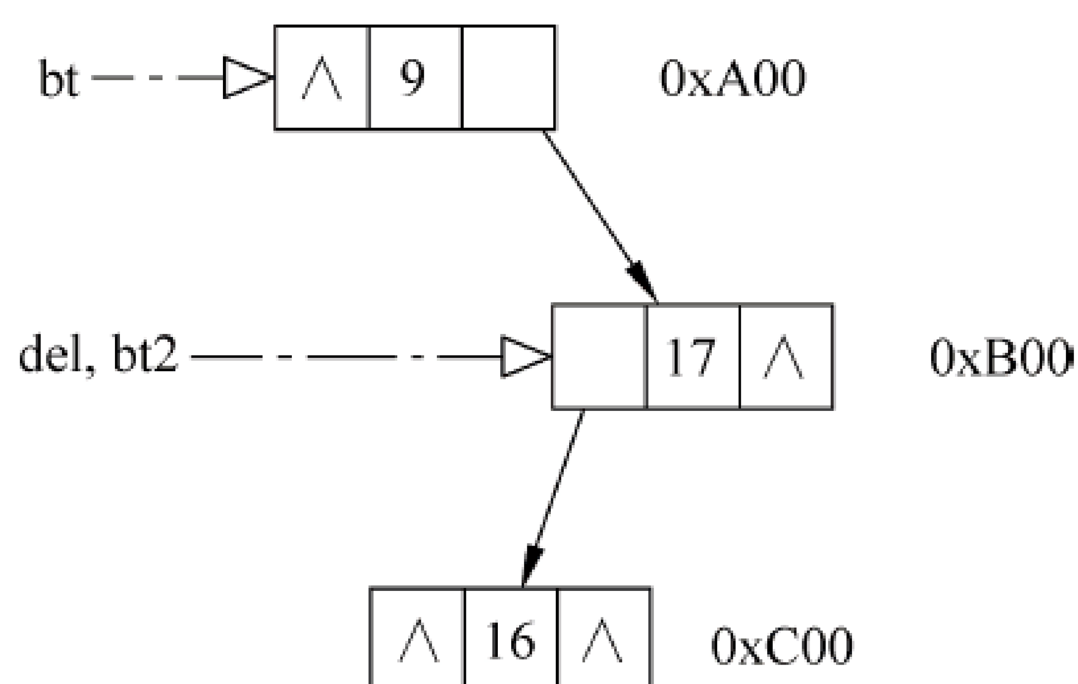


图 8.8　删除二叉排序树的特定结点

首先执行 DeleteBST(bt,17)⇒bt=0xA00 != NULL,bt—>key=9,k=17; k>bt→key 分支条件成立，调用 DeleteBST(b→rchild,17)。

其次执行 DeleteBST(bt2,17)(为了区分，将本次调用的形参 bt 用 bt2 表示)⇒ bt2=bt→rchild=0xB00,bt2→key=17,k=17; k==bt2→key 条件成立，调用 Delete(bt2)。

最后执行 Delete(del)⇒del=bt2=0xB00,del→rchild==NULL,tmp=del⇒tmp=0xB00,del=del→lchild; ⇒del=0xC00,free(tmp)⇒释放 tmp 所指结点(即 key 为 17 的结

点),返回到 Delete(bt2),将形参 del 回传给实参 bt2; bt2 = del = 0xC00,再返回到 DeleteBST(bt→rchild,17),将形参 bt2 回传给实参 bt→rchild⇒bt→rchild=bt2=0xC00,从而将根结点的右孩子指针指向 key 为 16 的结点,达到删除 key 为 17 的结点的目的。

8.3.2 平衡二叉树



视频讲解

虽然在二叉排序树上实现插入、删除和查找等基本操作的平均时间均为 $O(\log_2 n)$,但最坏情况下,这些基本运算的时间均会增至 O(n)。为了避免这种情况发生,人们研究了许多种动态平衡的方法,使得在树中插入或删除元素时,通过调整树的形态保持树的「平衡」,使之既保持 BST 性质不变,又保证树的高度在任何情况下均为 $O(\log_2 n)$,从而确保树上的基本运算在最坏情况下的时间也均为 $O(\log_2 n)$。平衡的二叉排序树有很多种,较著名的有 AVL 树。

若一棵二叉树中每个结点的左、右子树的高度至多相差 1,则称此二叉树为平衡二叉树。在算法中,通过**平衡因子**(Balance Factor,BF)具体实现上述平衡二叉树的定义。平衡因子的定义是:平衡二叉树中每个结点有一个平衡因子,每个结点的平衡因子是该结点左子树的高度减去右子树的高度。从平衡因子的角度说,若一棵二叉树中所有结点的平衡因子的绝对值 $|BF| \leqslant 1$,即平衡因子的值域为{1, 0, −1},则该二叉树为平衡二叉树。

【例 8.4】 图 8.9 是平衡二叉树和不平衡二叉树的例子。图中,结点旁标注的数字为该结点的平衡因子。其中,图 8.9(a)是一棵平衡二叉树,图中所有结点平衡因子的绝对值都小于等于 1;图 8.9(b)是一棵不平衡二叉树,图中结点 3 的平衡因子为−2。

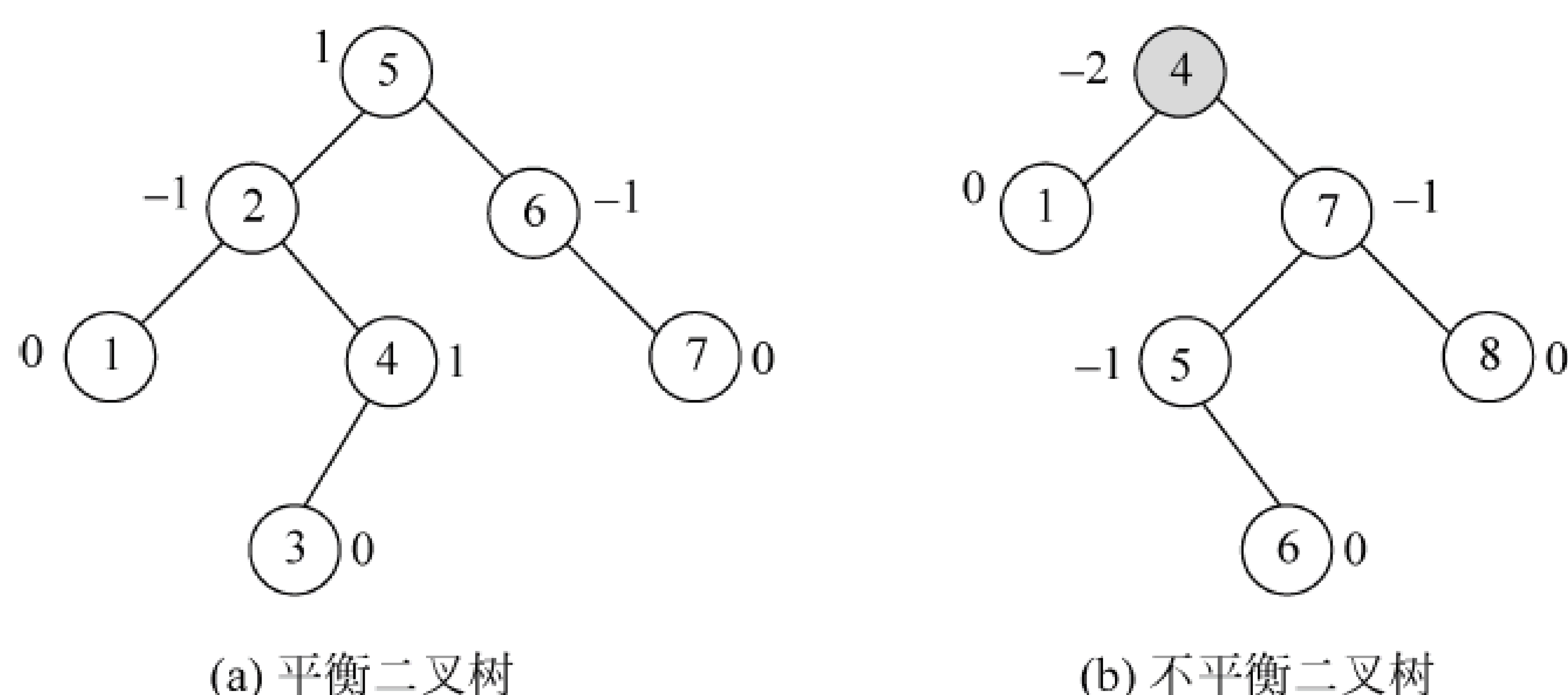


图 8.9 平衡二叉树和非平衡二叉树

如何使构造的二叉树是一棵平衡二叉树,而不仅仅是一棵二叉排序树,**关键**是每次向二叉插入新结点时要保持所有结点的平衡因子满足平衡二叉树的要求。这就要求一旦某些平衡因子在插入新结点后不满足要求,就要进行调整。

在讨论 AVL 树的基本运算算法前,定义其结点的类型如下。

```
typedef int KeyType;                    //定义关键字类型
typedef char InfoType;
typedef struct node {                   //记录类型
KeyType key;                            //关键字项
int bf;                                 //平衡因子
```

```
    InfoType data;                    //其他数据域
    struct node *lchild, *rchild;     //左、右孩子指针
} BSTNode;
```

1. 平衡二叉树插入结点的调整方法

若向平衡二叉树中插入一个新结点后破坏了平衡二叉树的平衡性，首先从该新插入结点向根结点方向找到第一个失去平衡的结点，然后以该失衡结点和与它相邻的下数两层的刚查找过的两个结点构成调整子树，使之成为新的平衡子树。当失去平衡的最小子树被调整为平衡子树后，原有的其他所有不平衡子树都无须调整，整棵二叉排序树又成为一棵平衡二叉树。

最小失衡子树是指以离插入结点最近，且平衡因子绝对值大于1的结点作为根的子树。

用A表示最小失衡子树的根结点，在下列图中，用长方框表示子树，用长方框的高度(并在长方框旁标有高度值h或h+1)表示子树的高度，用带阴影的小方框表示新插入的结点。调整子树的操作可归纳为下列4种情况。

1) LL型(左左型)调整——单顺

如图8.10所示，源自在A结点的左孩子(设为B结点)的左子树上插入新结点，使得A结点的平衡因子由1变成2，而引起的不平衡。调整的方法是：单次**顺时针旋转**调衡，即以线**A—B**作为旋转臂，以B作为转臂的轴心，顺时针旋转约90°后，于是结点A代替结点B的原右子树(β)，成为结点B的右子树的根结点；而B的原右子树(β)成为A的左子树。

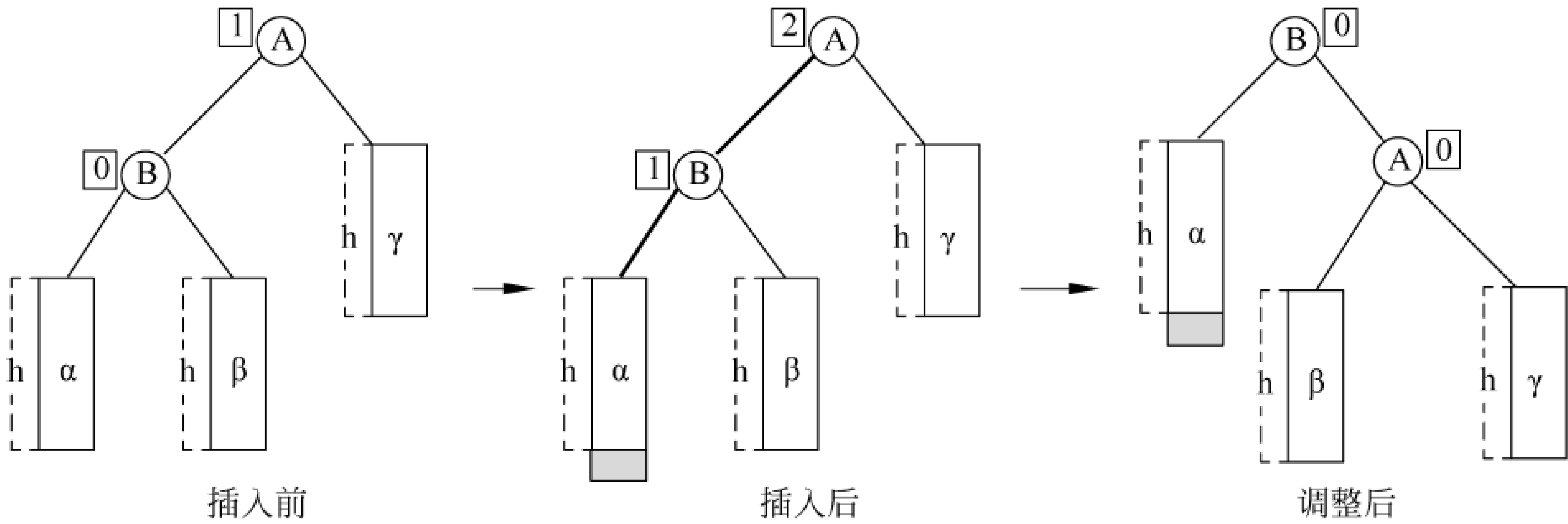


图8.10　LL型一般调整过程

2) RR型(右右型)调整——单逆

如图8.11所以，源自在A结点的右孩子(设为B结点)的左子树上插入新结点，使得A结点的平衡因子由−1变成−2，而引起的不平衡。调整的方法是：单次**逆时针旋转**调衡，即以线**A—B**作为旋转臂，以B作为转臂的轴心，逆时针旋转约90°后，于是结点A代替结点B的原左子树(β)，成为结点B的左子树的根结点；而B的原左子树(β)成为A的右子树。

3) LR型(左右型)调整——先逆后顺

如图8.12所示，源自在A结点的左孩子(设为B结点)的右子树上插入结点，使得A结点的平衡因子由1变成2，而引起的不平衡。调整的方法是：先**逆时针**旋转，后**顺时针**旋转调衡，即

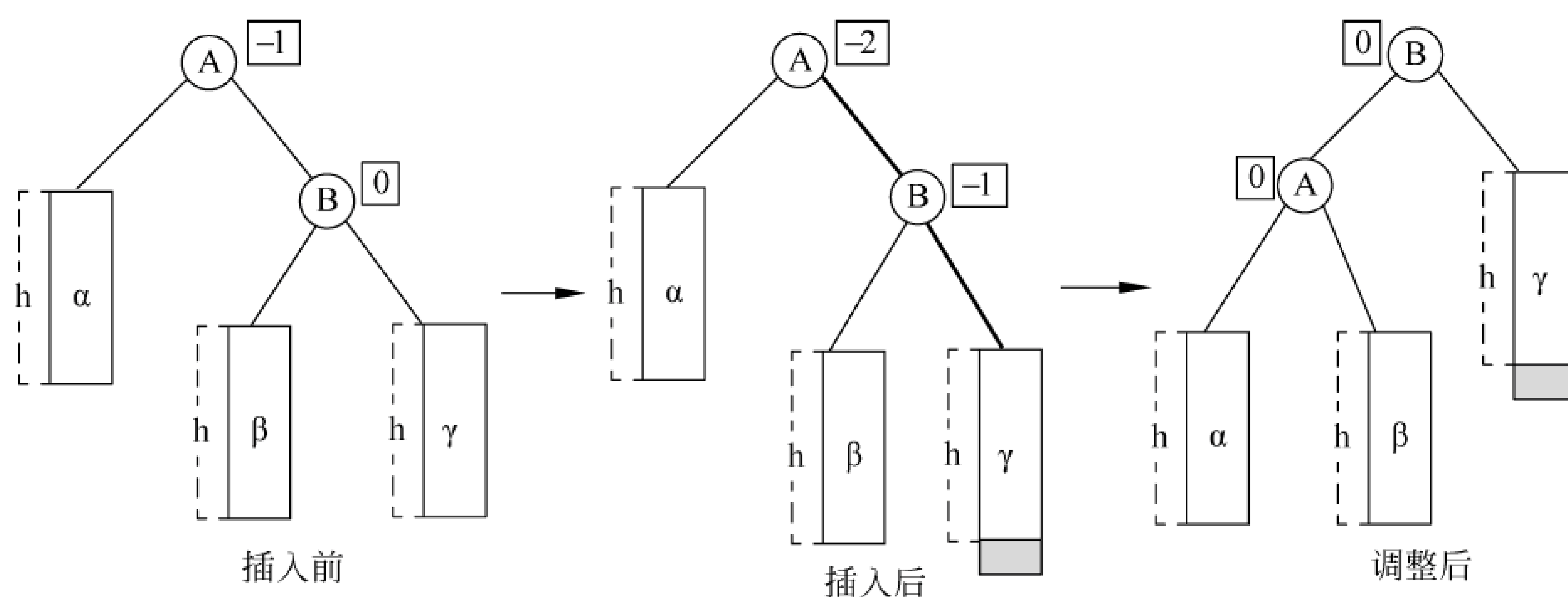


图 8.11　RR 型一般调整过程

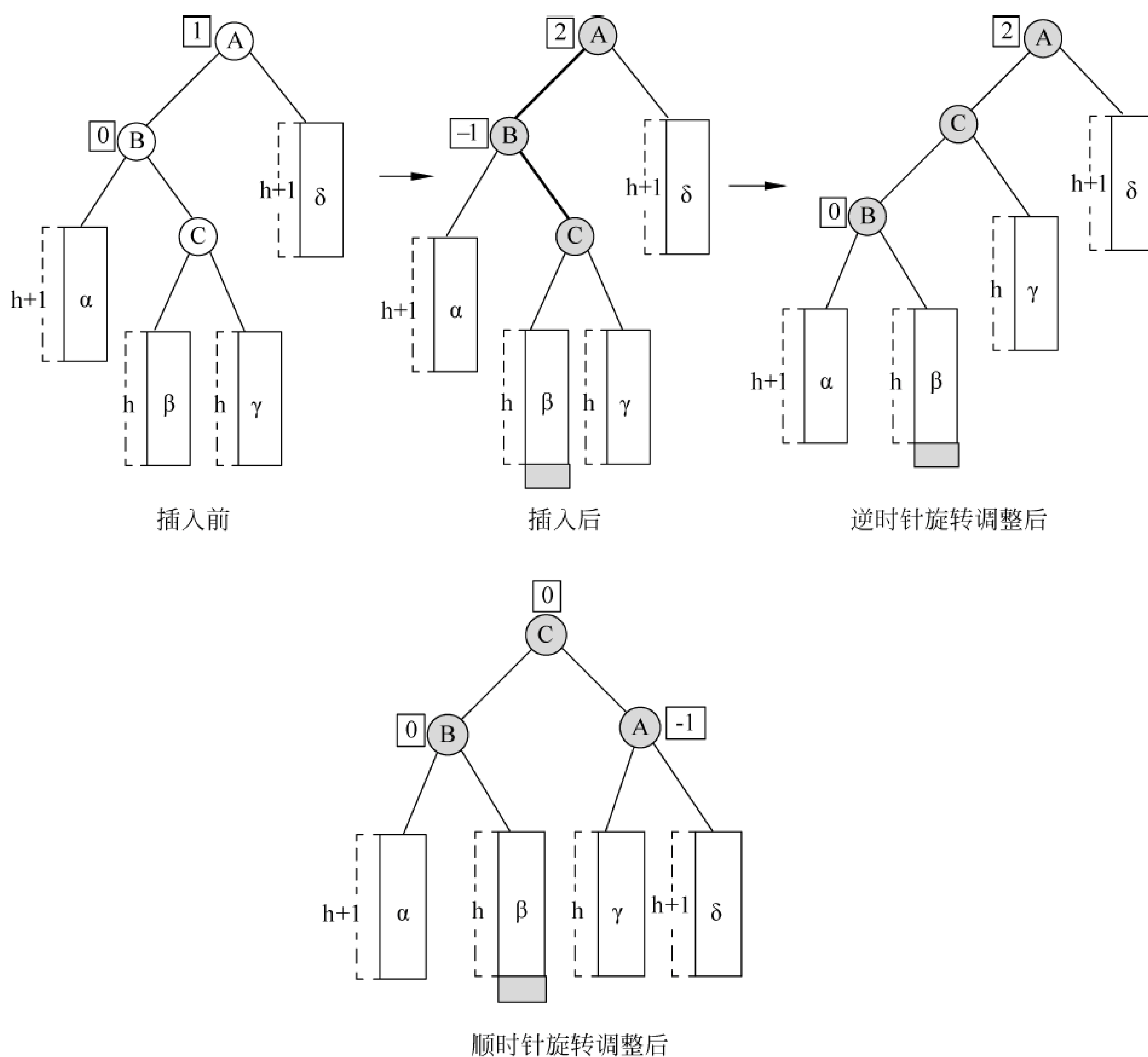


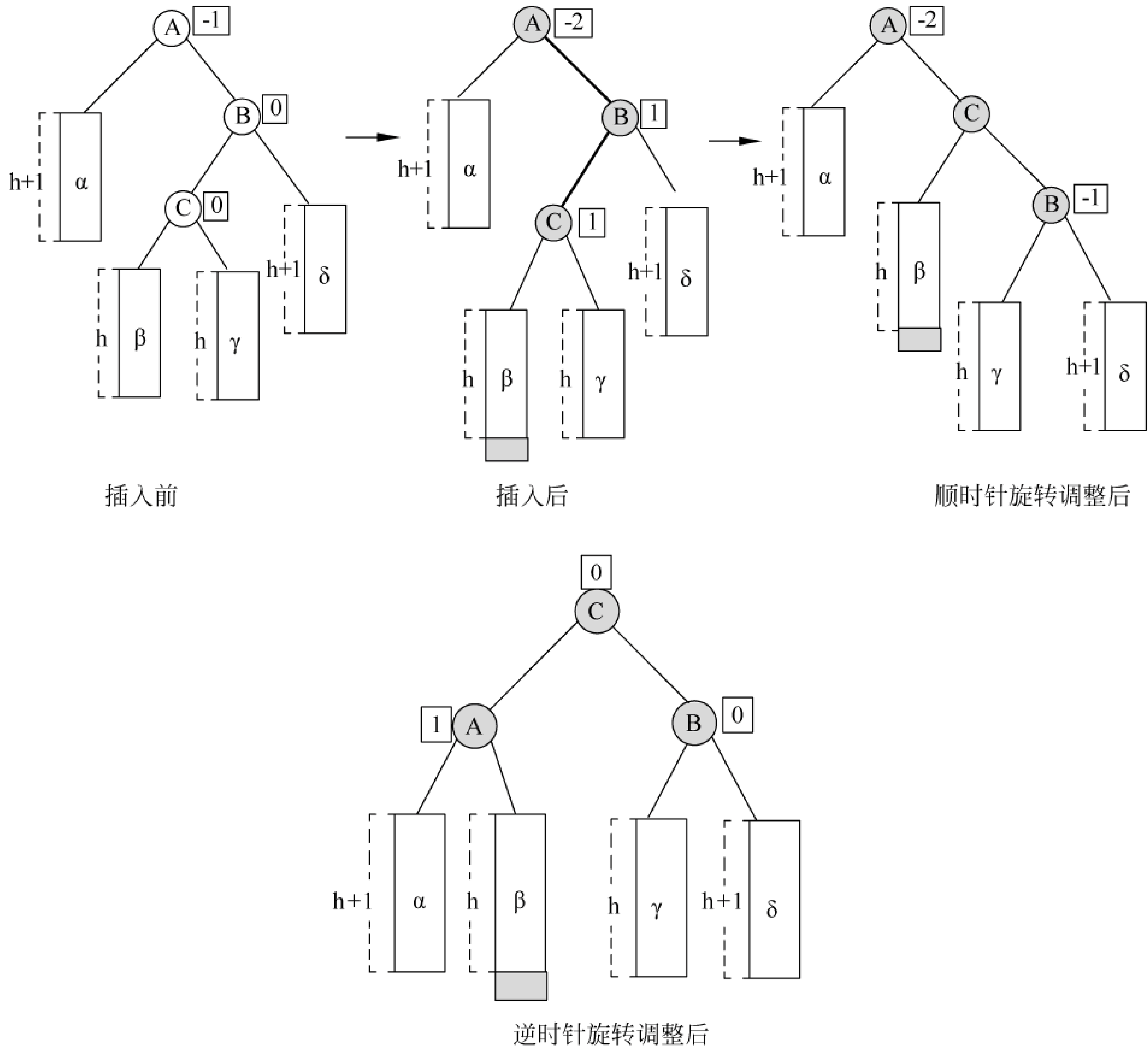
图 8.12　LR 型一般调整过程

以线 **B—C** 作为旋转臂,以 C 作为转臂的轴心,**逆时针旋转**约 90°后,于是结点 B 代替结点 C 的原左子树(β),成为结点 C 的左子树的根结点;而 C 的原左子树(β)则成为 B 的右子树。此刻,A,B,C 结点形成一条直线,失衡问题并未解决。

以线 **A—C** 作为旋转臂,以 C 作为转臂的轴心,**顺时针旋转**约 90°后,于是结点 A 代替结点 C 的原右子树(γ),成为结点 C 的右子树的根结点;而 C 的原右子树(γ)则成为 A 的左子树。

4）RL 型（右左型）调整——先顺后逆

如图 8.13 所示，源自在 A 结点的右孩子（设为 B 结点）的左子树上插入结点，使得 A 结点的平衡因子由 −1 变成 −2，而引起的不平衡。调整的方法是：先**顺时针**旋转，后**逆时针**旋转调衡，即



0
C
1 A
B 0
h+1 α
h β
h γ
h+1 δ
逆时针旋转调整后

图 8.13　RL 型一般调整过程

以线 **B—C** 作为旋转臂，以 C 作为转臂的轴心，**顺时针旋转**约 90°后，于是结点 B 代替结点 C 的原右子树（γ），成为结点 C 的右子树的根结点；而 C 的原右子树（γ）则成为 B 的左子树。此刻，A，B，C 结点形成一条直线，失衡问题并未解决。

以线 **A—C** 作为旋转臂，以 C 作为转臂的轴心，**逆时针旋转**约 90°后，于是结点 A 代替结点 C 的原左子树（β），成为结点 C 的左子树的根结点；而 B 的原左子树（β）则成为 A 的右子树。

下面介绍等价二叉排序树的证明。

若两棵二叉排序树的中序序列相同，则称它们相互等价；反之亦然。

易证：利用上述 4 种方法调整，调整前后对应的中序序列均相同，换言之，彼此等价，因此，调整后仍保持二叉排序树的性质不变。

由同一组共 8 个结点组成，相互等价的两棵二叉排序树（拓扑差异，以阴影标出）如图 8.14 所示。

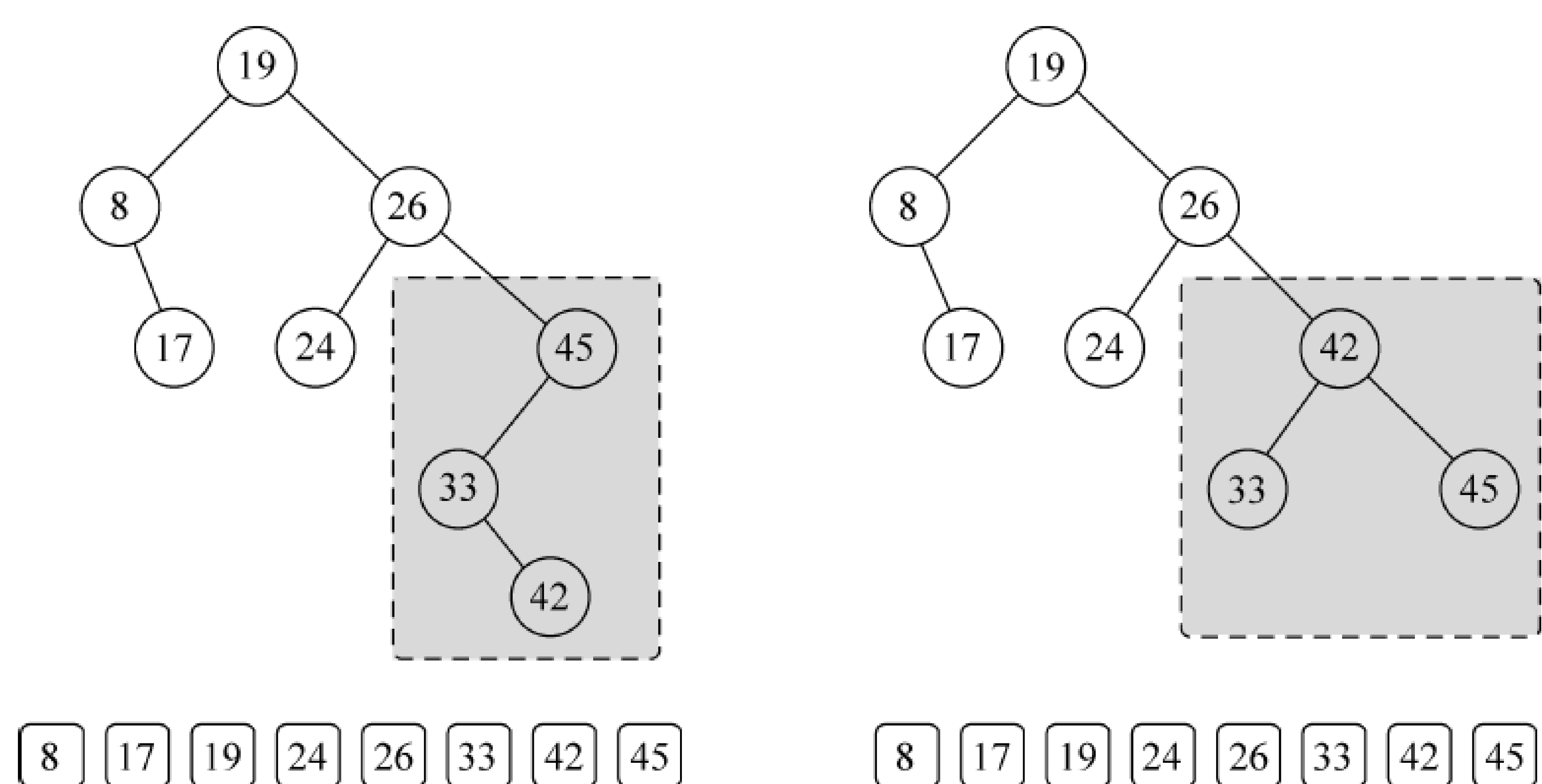


图 8.14　由同一组共 8 个结点组成，相互等价的两棵二叉排序树(拓扑差异，以阴影标出)

【例 8.5】 输入关键字序列{17,4,6,12,8,27,19,15}，给出构造一棵 AVL 树的过程。

解：建立 AVL 树的过程如图 8.15 所示。

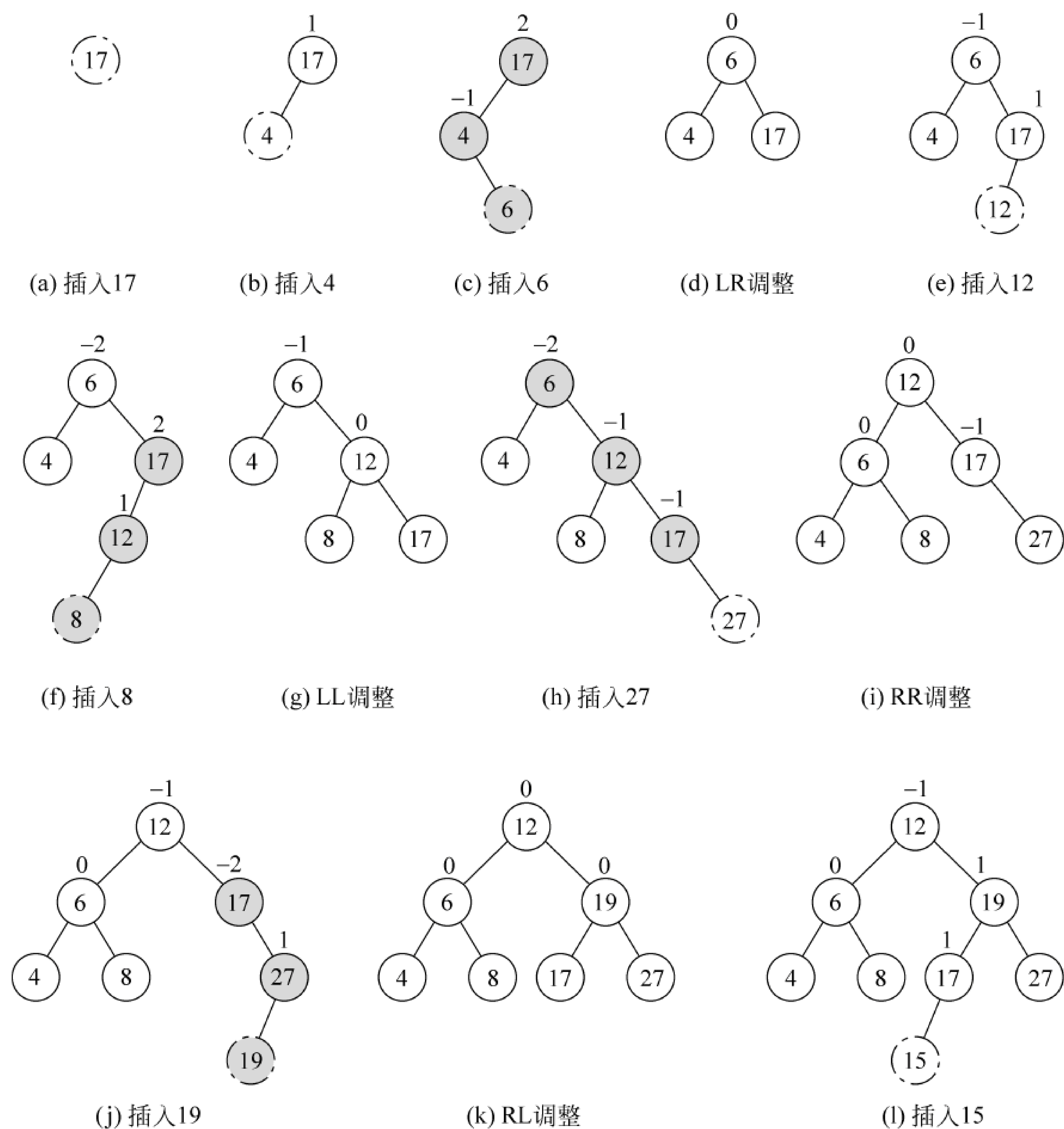


图 8.15　建立 AVL 树的过程

注意：进行重平衡调整前，应先判断出其所属的失衡类型——先识别出最小失衡子树之根 A，而后沿着插入路径向下数两层，分别标为 B，C，此时根据连接 A，B，C 三结点的边的走向，便可判定失衡类型为{LL，RR，LR，RL}中的哪一种。

2. 平衡二叉树删除结点的调整方法

平衡二叉树的删除结点操作与插入操作有颇多相似之处。

在平衡二叉树上删除结点 D(假定有且仅有一个结点值等于 D)的过程如下。

step1：采用二叉排序树的删除方法——找到结点 D 并删除之。

step2：沿根结点到被删除结点的路线之逆，逐层向上查找，必要时修改 D 祖先结点的平衡因子，因为删除结点 D 后，可能会使某些子树的高度降低。

step3：查找途中一旦发现 D 的某个祖先 * p 失衡，就要进行调整，具体判定规则如下。

- 假设结点 D 在 * p 的左子树中，在结点 * p 失衡后做何种调整取决于 * p 右孩子 * pr。
  若 * pr 的平衡因子是 1，说明它的左子树高，需做 RL 调整。
  若 * pr 的平衡因子是－1，说明它的右子树高，需做 RR 调整。
  若 * pr 的平衡因子是 0，则做 RL 或 RR 调整均可。
- 如果结点 D 在 * p 的右子树中，调整过程类似，但取决于 * p 左孩子 * pl。
  若 * pl 的平衡因子是 1，说明它的左子树高，需做 LL 调整。
  若 * pl 的平衡因子是－1，说明它的右子树高，需做 LR 调整。
  若 * pl 的平衡因子是 0，则做 LR 或 LL 调整均可。

step4：如果调整之后，对应的子树的高度降低了，这个过程还将继续，直到根结点为止。换言之，在平衡二叉树上删除一个结点有可能引起多次调整，不像插入结点那样至多调整一次。

【例 8.6】 对例 8.5 生成的 AVL 树，给出删除结点{12，8，17}的步骤。

解：删除 AVL 中结点的过程如图 8.16 所示(其中，图 8.16(a)为初始 AVL 树)。

step1：删除值为 12 的结点(根结点)时，先让根结点和左子树最大结点(值 8)的数据域交换，将问题化简为——删除左子树最大结点(其必为最右下结点，单左分支结点易删)，找到其双亲结点(值 6)，修改它的平衡因子为 1，再向上找到根结点，此时它已为平衡，如图 8.16(b)所示。

step2：删除结点 8(为根结点)时，同理，互换根结点和左子树最大结点 6 的数据域，删除左子树最大结点，其双亲结点值现为 6(原为结点值 8)，修改它的平衡因子为－2，不平衡——找其右孩子结点(值 19)，修改它的平衡因子为 1，根据规则知需作 RL 调整，如图 8.16(d)所示。

step3：删除结点 17(为根结点)时，同上述过程，互换，删除左子树最大结点，找到其双亲结点(值 6)，修改它的平衡因子为 1，再向上找到根结点，此时它已为平衡，如图 8.16(e)所示。

3. 平衡二叉树的查找

在平衡二叉树上进行查找的过程和在二叉排序树上进行查找的过程完全相同，因此，在平衡二叉树上进行查找时，关键字的比较次数不会超过平衡二叉树的深度。在最坏的情况下，普通二叉排序树的查找长度为 O(n)。对于平衡二叉树来说，含有 n 个结点的平衡二叉树的平均查找长度为 $O(\log_2 n)$。

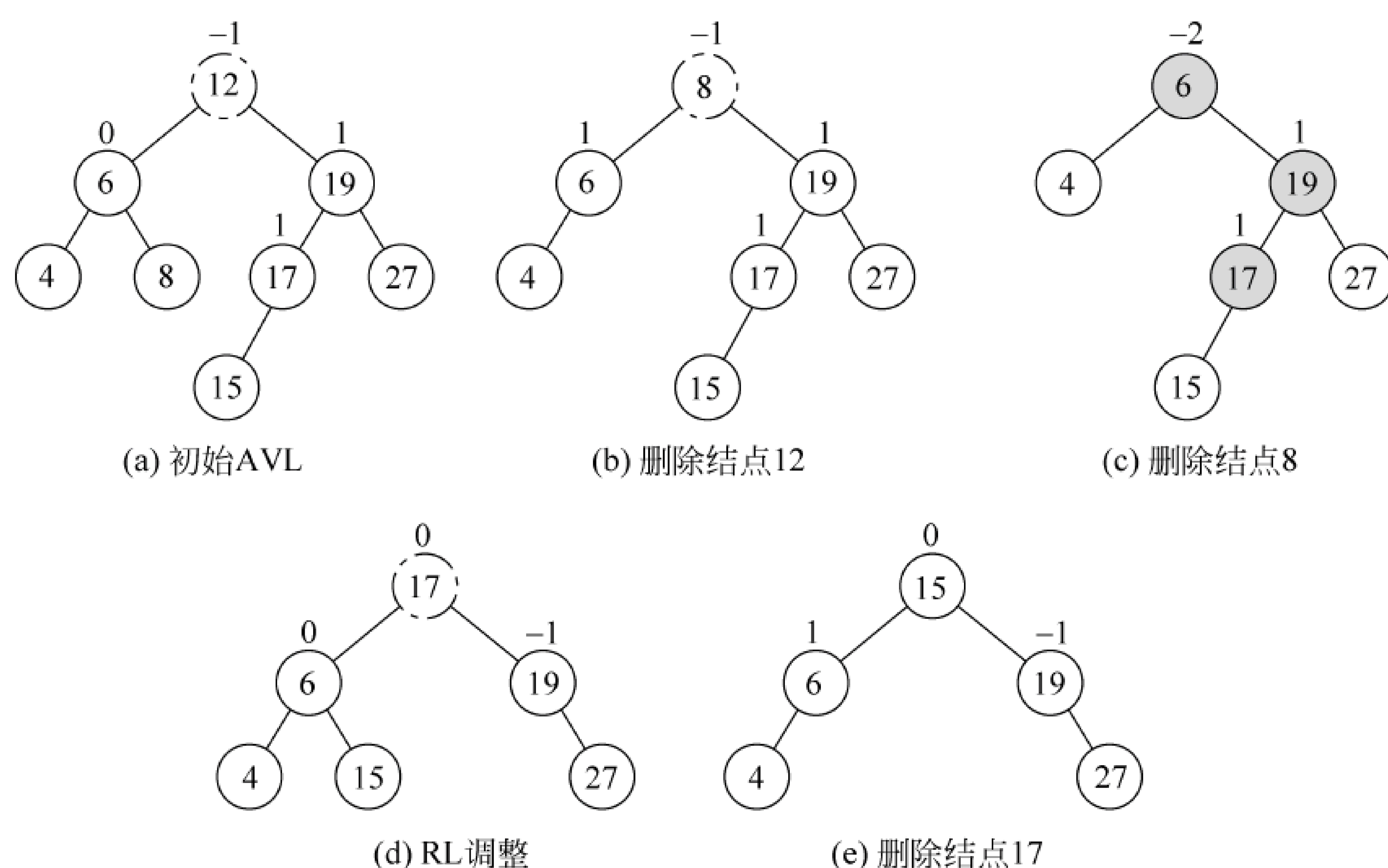


图 8.16 删除 AVL 中结点的过程

8.3.3 B－树


视频讲解

BST 和 AVL 树都是用作内部查找的数据结构,即查找的数据集不大,可以放在内存中。本小节介绍的 B－树和 8.3.4 小节介绍的 B＋树是用作外部查找的数据结构,其中的数据存放在**外存**中。

B－树中所有结点的孩子结点数的最大值称为 B－树的**阶**,通常用 m 表示,从查找效率考虑,要求 m≥3。一棵 m 阶 B－树要么是一棵空树,要么是满足下列要求的 m 叉树。

- 所有的叶子结点在同一层,并且不带信息。
- 树中每个结点至多有 m 棵子树(至多含有 m－1 个关键字)。
- 若根结点不是终端结点,则根结点至少有两棵子树。
- 除根结点外,其他**非叶子结点**至少有 $\left\lceil \frac{m}{2} \right\rceil$ 棵子树 $\left(\text{即至少含有}\left\lceil \frac{m}{2} \right\rceil -1\text{ 个关键字}\right)$。
- 每个非叶子结点的结构为

n	p_0	k_1	p_1	k_2	…	k_n	p_n

其中各个字符的意义如下。

- n 为该结点中的关键字个数,除根结点外,其他所有非叶子结点的关键字个数 n 都满足:$\lceil m/2 \rceil -1 \leqslant n \leqslant m-1$。
- $k_i(1<i<n)$为该结点的关键字且满足 $k_i<k_{i+1}$。
- $p_i(0\leqslant i\leqslant n)$为该结点的孩子结点指针且 $p_i(0\leqslant i\leqslant n-1)$所指结点上的关键字 $k_i \leqslant k_{pi} < k_{i+1}$,特别地,最后一个指针 p_n 所指结点上的关键字大于 k_n。

【例 8.7】 图 8.17 是一棵 3 阶 B－树,m＝3。它分别满足上述对应的要求:

- 所有叶子结点都在同一层上。

• 每个结点的孩子个数小于等于 3，即拥有的子树不超过 m(m=3)棵。
• 根结点为非终端结点，必有两个孩子结点。
• 除根结点外，非叶子结点至少有⌈m/2⌉=⌈1.5⌉=2 个孩子，至少有⌈m/2⌉−1=1 个关键字。
• 除根结点外的所有非叶子结点的关键字个数 n，有⌈m/2⌉−1=1≤n≤m−1=2。

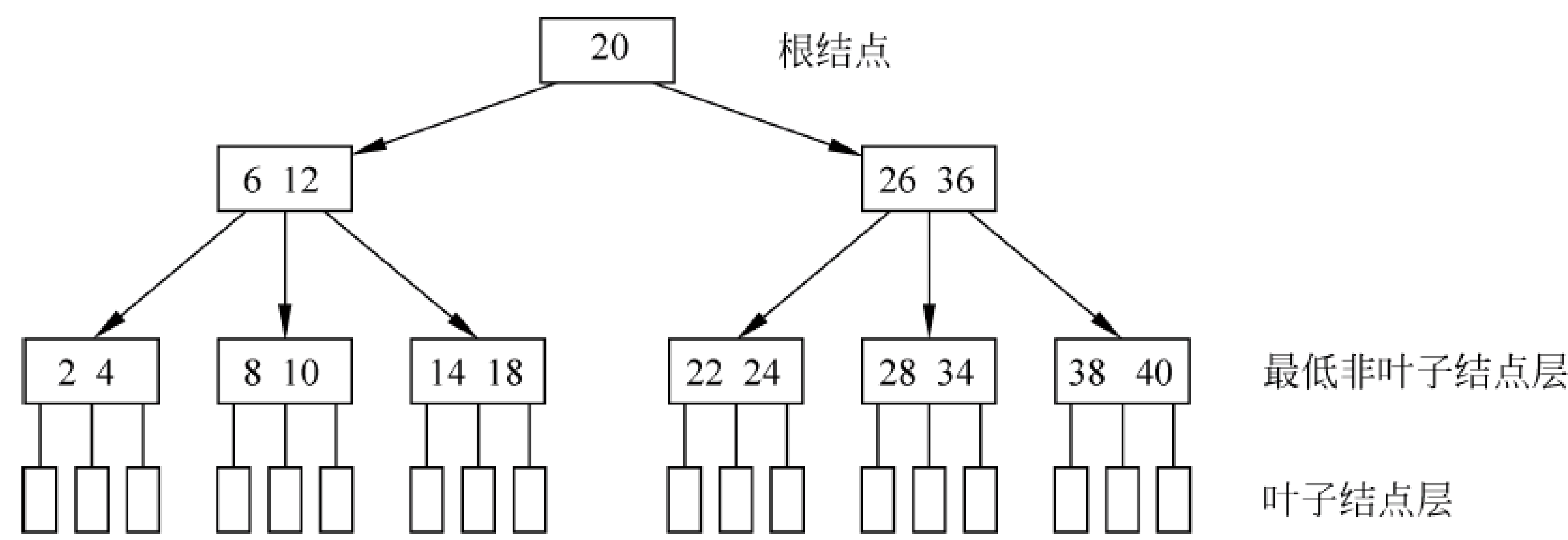


图 8.17　一棵 3 阶 B−树

在 B−树中，叶子结点不带信息(可看做是外部结点或查找失败的结点，实际上，这些结点不存在，指向这些结点的指针均为空)。

注意：为了直观起见，后面的 B−树的图中都没有画出叶子结点层。

在 B−树的存储结构中，各个结点的类型定义如下。

```
#define MAXM 10                    //定义 B−树的最大的阶数
typedef int KeyType;               //KeyType 为关键字类型
typedef struct node                //B−树结点类型的定义
{ int keynum;                      //结点当前拥有的关键字的个数
  KeyType key[MAXM];               //key[1..keynum]存放关键字,key[0]不用
  struct node *parent;             //双亲结点指针
  struct node *ptr[MAXM];          //孩子结点指针数组 ptr[0..keynum]
} BTNode;
int m;                             //m 阶 B−树,为全局变量
int Max;                           //m 阶 B−树中每个结点的至多关键字个数,Max=m−1
int Min;                           //m 阶 B−树中非叶子结点的至少关键字个数,Min=(m−1)/2
```

为了方便在 B−树查找时返回结果，定义如下类型。

```
typedef struct
{
 BTNode *pt;                         //指向找到的结点
 int i;                              //1..m, 在结点中的关键字序号
 int tag;                            //1:查找成功, 0:查找失败
} Result;                            //B−树的查找结果类型
```

当查找的返回值 tag 为 0 时，表示查找失败；当 tag 为 1 时，表示查找的结果为结点 *pt 的 key[i]关键字。

1. B－树的查找

在B－树中查找给定关键字的方法类似于二叉排序树上的查找。不同的是,在每个结点上确定向下查找的路径不一定是二路的,而是n+1路的,因此也称之为多路平衡查找/搜索树。因为结点内的关键字序列key[1..n]有序,故既可以用顺序查找,也可以用折半查找。

在一棵B－树上查找关键字为k的方法为:

将k与根结点中的key[i]进行比较,直到找完路径上的全部待查结点。

- 若k=key[i],则查找成功。
- 若k < key[1],则沿着指针ptr[0]所指的子树继续查找。
- 若key[i] < k < key[i+1],则沿着指针ptr[i]所指的子树继续查找。
- 若k > key[n],则沿着指针ptr[n]所指的子树继续查找。
- 若指向待查结点的指针为NULL且found标记为0,则表示查找失败。

对应的查找算法如下。

/** 在m阶B－树t上查找关键字k,返回结果(pt,i,tag)。

* 若查找成功,则特征值tag=1,指针pt所指结点中第i个关键字等于k。

* 否则,特征值tag=0,关键字k的插入点应在指针Pt所指结点中第i个和第i+1个关键字之间。

*/

```
Result SearchBTree(BTNode *t, KeyType k)
{
BTNode *p = t, *q = NULL;                    //初始化,p指向待查结点,q指向p的双亲
int found = 0, i = 0;
Result r;
while (p != NULL && found == 0)
{
  i = Search(p, k);
      //在 p->key[1...keynum] 中查找i,使得 p->key[i] <= k< p->key[i+1]
  if (i>0 && p->key[i] == k)                 //找到待查关键字
      found=1;
  else
{
      q = p;
      p = p->ptr[i];
  }
}
  r.i = i;
  if (found==1)
{                                            //查找成功
    r.pt=p;r.tag=1; //指针pt所指结点中第i个关键字等于k,两个须联立解释
}
  else
{                                            //查找不成功,返回k的插入位置信息
    r.pt=q;r.tag=0;
```

```
    }
    return r;                          //返回 k 的位置(或插入位置)
}
int Search(BTNode *p,KeyType k)
{
 //在 p->key[1..keynum]中查找 i,使得 p->key[i]<=k<p->key[i+1]
 int i = 0;
 for( ;i<p->keynum && p->key[i+1]<=k;i++)
    ;
 return i;
}
```

在 B—树中进行查找时,其查找时间主要花费在搜索结点上,即主要取决于 B—树的**深度**。那么,含有 n 个关键字的 m 阶 B—树可能达到的最大深度 h 为多少呢? 或者说,深度为 h 的 B—树中至少含有多少个结点?

- 第 1 层最少结点数为 1 个。
- 第 2 层最少结点数为 2 个。
- 第 3 层最少结点数为 $2\lceil m/2\rceil$个。
- 第 4 层最少结点数为 $2\lceil m/2\rceil^2$个。
- ……
- 第 h+1 层最少结点数为 $2\lceil m/2\rceil^{h-1}$个。(一般化)

假设 m 阶 B—树的深度为 h+1,由于第 h+1 层为叶子结点,而当前树中含有 n 个关键字,则叶子结点数必为 n+1 个,由此可推得下列结果:

- $n+1\geqslant 2\left\lceil\frac{m}{2}\right\rceil^{h-1}$
- $h-1\leqslant \log_{\lceil\frac{m}{2}\rceil}\frac{(n+1)}{2}$
- $h\leqslant \log_{\lceil\frac{m}{2}\rceil}\frac{(n+1)}{2}+1$

因此,在含 n 个关键字的 B—树上进行查找,需访问的结点个数不超过$\log_{\lceil\frac{m}{2}\rceil}\frac{(n+1)}{2}+1$个。也就是说,在含 n 个关键字的 B—树上查找的时间复杂度为 $O\left(\log_{\lceil\frac{m}{2}\rceil}\frac{(n+1)}{2}+1\right)$。

2. B—树的插入

将关键字 k 插入到 B—树的过程分两步完成。

step1:利用前述的 B—树的查找算法找出该关键字的插入结点(**注意**,B—树的插入结点一定属于最低非叶子结点层)。

step2:判断该结点是否还有空位置,即判断该结点是否满足 n<m−1:

- 若该结点满足 n<m−1,则说明该结点还有空位置,直接把关键字 k 插入到该结点的合适位置上(即满足插入后结点上的关键字仍保持有序)。
- 若有 n=m−1,则说明该结点已没有空位置,需要把结点**分裂**成两个。

当目标结点已没有空位置存放新插关键字时,结点须进行的**分裂**操作如下。

step1：取一新结点，把原结点上的关键字和 k 按升序排列。

step2：从中间位置(即⌈m/2⌉处)把关键字(不包括中间位置的关键字)分成两部分——左部分所含关键字放在旧结点中，右部分所含关键字放在新结点中。

step3：中间位置的关键字连同新结点的存储位置插入到双亲结点中。

注意：如果双亲结点的关键字个数也超过 Max，则要再分裂，再往上插，直至这个过程传递到根结点为止(向上传递)。B—树的生成就是从一棵空树开始，逐个插入关键字。

【例 8.8】 关键字序列为{2，4，12，14，22，8，16，26，20，10，34，18，32，40，6，24，28，36，38，30}，创建一棵 5 阶 B—树。

解：一棵 5 阶 B—树的建立过程如图 8.18 所示(其中，阴影六角星表示致使分裂的新插关键字)。

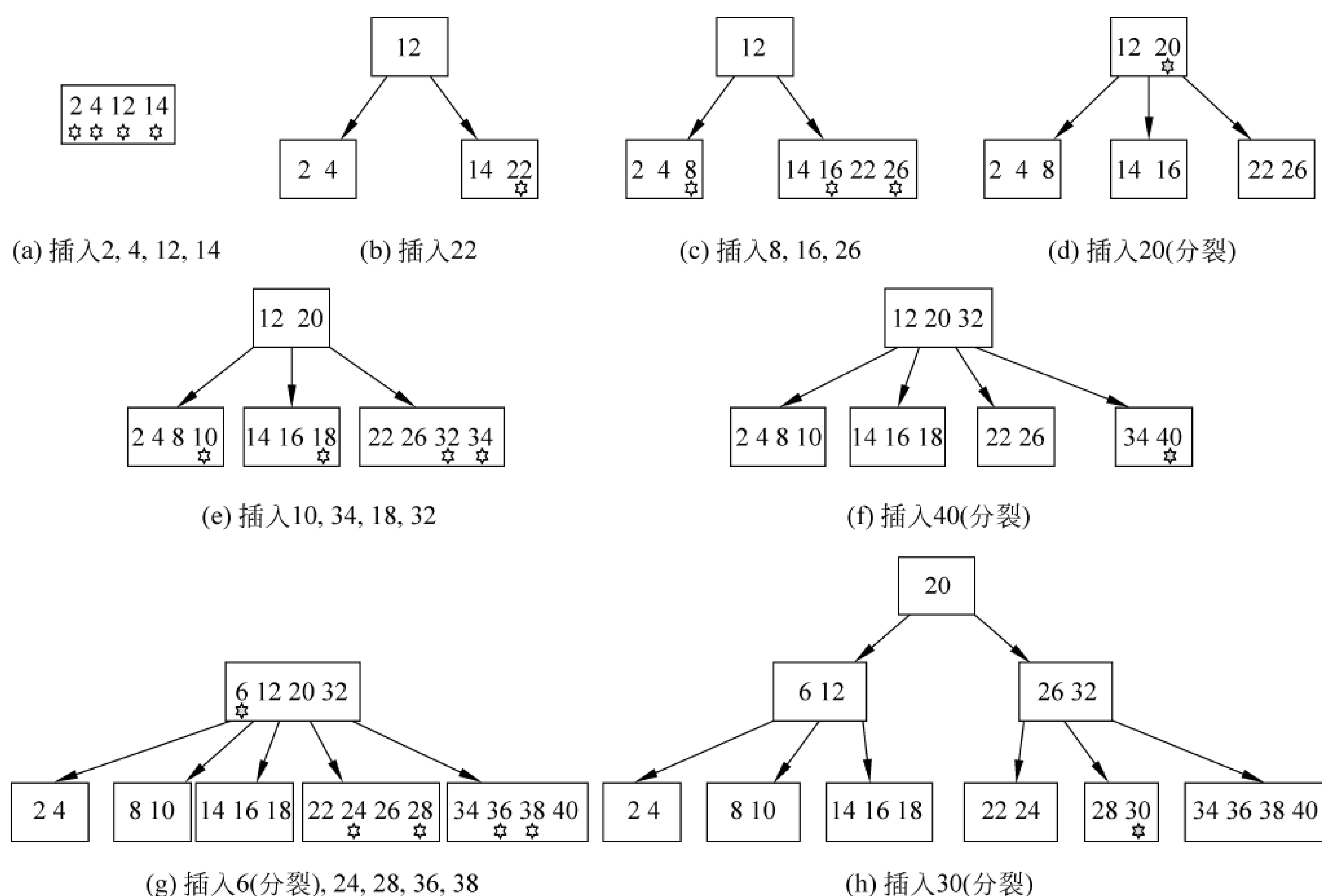


图 8.18　一棵 5 阶 B—树的建立过程

由于 m = 5，所以每个结点的关键字个数为 2～4。以图 8.18(a)～(h)为例说明插入过程。在图 8.18(c)中插入关键字 20 时，查到其位置应在最右边的结点中间，即该结点变成{14，16，20，22，26}，关键字个数不符合要求，需进行分裂，即该结点变成两个结点，分别包含关键字{14，16}和{22，26}，并将中间关键字 20 移至双亲结点中，双亲结点变为 12，20。其他分裂过程可做类似分析，此处不一一详解，留给读者自己练习。

3. B—树的删除

B—树的删除过程与插入过程类似，只是更为复杂一些。在删除某结点中的某一关键字后，该结点中的余下关键字个数 n 需满足 n≥⌈m/2⌉−1，如若不然，则将涉及结点的「合并」问题。在 B—树上删除关键字 k 的过程也分两步完成。

step1：利用前述的 B—树的查找算法，找出该关键字所在的结点。

step2：在结点上删除关键字 k，分两种情况。

一种是在最低非叶子结点层的结点上删除关键字；共有以下 3 种具体情况。

- 假如被删结点的关键字个数大于 min（$=\lceil m/2 \rceil$），说明删去该关键字后对应结点仍满足 B—树的定义，则可直接删去该关键字。
- 假如被删结点的关键字个数等于 min，说明删去该关键字后对应结点将不满足 B—树的定义。此时若该结点的左（或右）兄弟结点中的关键字个数大于 min，则把该结点的左（或右）兄弟结点中最大（或最小）的关键字上移到双亲结点中，同时把双亲结点中大于（或小于）上移关键字的关键字下移到要删除关键字的结点中，这样删去关键字 k 后，该结点以及它的左（或右）兄弟结点都仍旧满足 B—树的定义。
- 假如被删结点的关键字个数等于 min，并且该结点的左和右兄弟结点（如果存在）中关键字个数均等于 min。这时需把要删除关键字的结点与其左（或右）兄弟结点以及双亲结点中分割二者的关键字**合并**成一个结点。

如果合并操作使双亲结点中关键字个数小于 min，则对双亲结点做同样处理，重复过程，直到根结点为止，可能使整个树减少一层。

另一种是在其他非叶子结点上删除关键字。过程如下。

- 假设要删除关键字 key[i]（1＜i＜n），在删去该关键字后，以该结点 ptr[i]所指子树中的最小关键字 key[min]代替被删关键字。

注意：ptr[i]所指子树中的最小关键字 key[min]一定在叶子结点上。

- 再以指针 ptr[i]所指结点为根结点查找并删除 key[min]（即以 ptr[i]所指结点为 B—树的根结点，以 key[min]为要删除的关键字，再次调用 B—树上的删除算法）。

转化：这样也就把在其他非叶子结点上删除关键字 k 的问题转换成在最低非叶子结点层的结点上删除关键字 key[min]的问题了。

【例 8.9】 对于生成的 B—树，给出删除 16，32，30，8 这 4 个关键字的过程。

解：在一棵 5 阶 B—树上删除关键字的过程如图 8.19 所示。

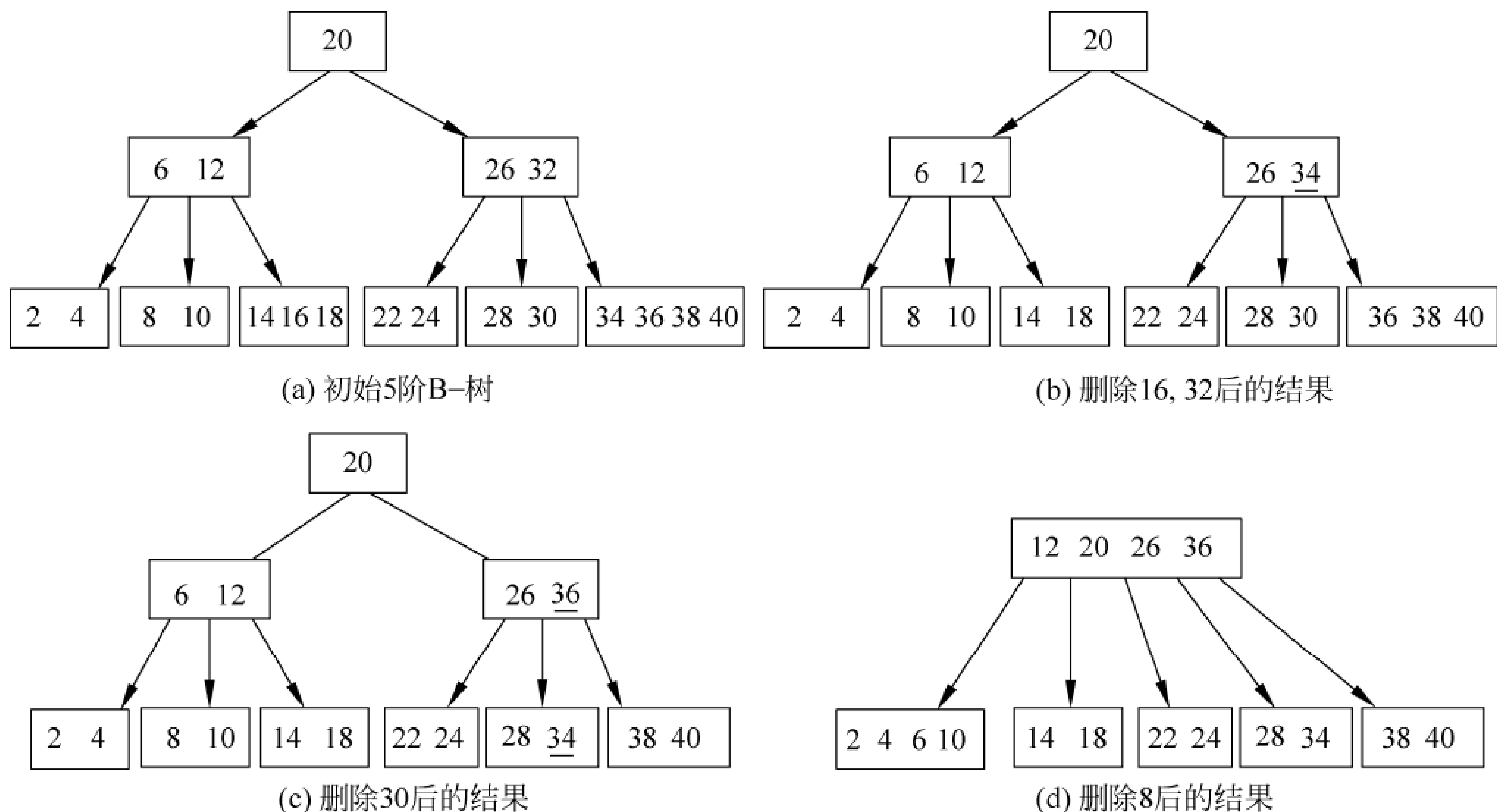


图 8.19 在一棵 5 阶 B—树上删除关键字的过程

由于 m=5，所以每个结点的关键字个数 n 最少为⌈m/2⌉−1=⌈2.5⌉−1=2。以图 8.19(c)～(d)为例说明删除过程。在图 8.19(c)中，删除关键字 8，使得包含它的叶子结点只有一个关键字，而其左、右兄弟结点都只有 2 个关键字，无法「外借」，需将它与左结点以及双亲结点「**合并**」。合并后，{2,4}结点变成{2,4,6,10}，双亲结点变成{12}，又不满足 B−树的条件，还需将双亲结点与其右结点以及根结点「**合并**」，即将 12 和 20 移到右兄弟结点{26,36}中，变成{12,20,26,36}，而根结点变空，故 B−树减少一层。

8.3.4　B+树

在索引文件组织中经常使用 B−树的一些变形，其中 B+树是一种应用广泛的变形。一棵 m 阶 B+树须满足下列条件。

- 每个分支结点至多有 m 棵子树。
- 根结点或者没有子树，或者至少有两棵子树。
- 除根结点外，其他每个分支结点至少有⌈m/2⌉棵子树。
- 有 n 棵子树的结点有 n 个关键字。
- 所有叶子结点都包含全部关键字及指向相应记录的指针，而且叶子结点按关键字大小顺序链接(可以把每个叶子结点看成是一个基本索引块，它的指针不再指向另一级索引块，而是直接指向数据文件中的记录)。
- 所有分支结点(可看成是索引的索引)中仅包含它的各个子结点(即下级索引的索引块)中最大关键字及指向子结点的指针。

注意：m 阶的 B+树和 m 阶的 B−树的主要**差异**如下。

- 在 B+树中，具有 n 个关键字的结点含有 n 棵子树，即每个关键字对应一棵子树，而在 B−树中，具有 n 个关键字的结点含有(n+1)棵子树。
- 在 B+树中，每个结点(除根结点外)中的关键字个数 n 的取值范围是⌈m/2⌉≤n≤m，根结点 n 的取值范围是 2≤n≤m；而在 B−树中，除根结点外，其他所有非叶子结点的关键字个数 n 的取值范围为⌈m/2⌉−1≤n≤m−1，根结点关键字个数 n 的取值范围为 1≤n≤m−1。
- B+树中的所有叶子结点 t 包含了全部关键字，即其他非叶子结点中的关键字包含在叶子结点中，而在 B−树中，关键字是不重复的。
- B+树中所有非叶子结点仅起到索引的作用，即结点中的每个索引项只含有对应子树的最大关键字和指向该子树的指针，不含有该关键字对应记录的存储地址。而在 B−树中，每个关键字对应一个记录的存储地址。
- 通常在 B+树上有两个头指针：一个指向根结点，另一个指向关键字最小的叶子结点，所有叶子结点链接成一个不定长的线性链表。

【例 8.10】 图 8.20 所示为一棵 4 阶的 B+树，其中叶子结点的每个关键字下面的指针指向对应记录的存储位置。通常在 B+树上有两个头指针：一个指向根结点，这里为 root；另一个指向关键字最小的叶子结点，这里为 sqt。

1. B+树的查找

在 B+树中可以采用两种查找方式：一种是直接从最小关键字开始进行顺序查找；另一种是从 B+树的根结点开始进行随机查找。后一种查找方式与 B−树的查找方式相似，

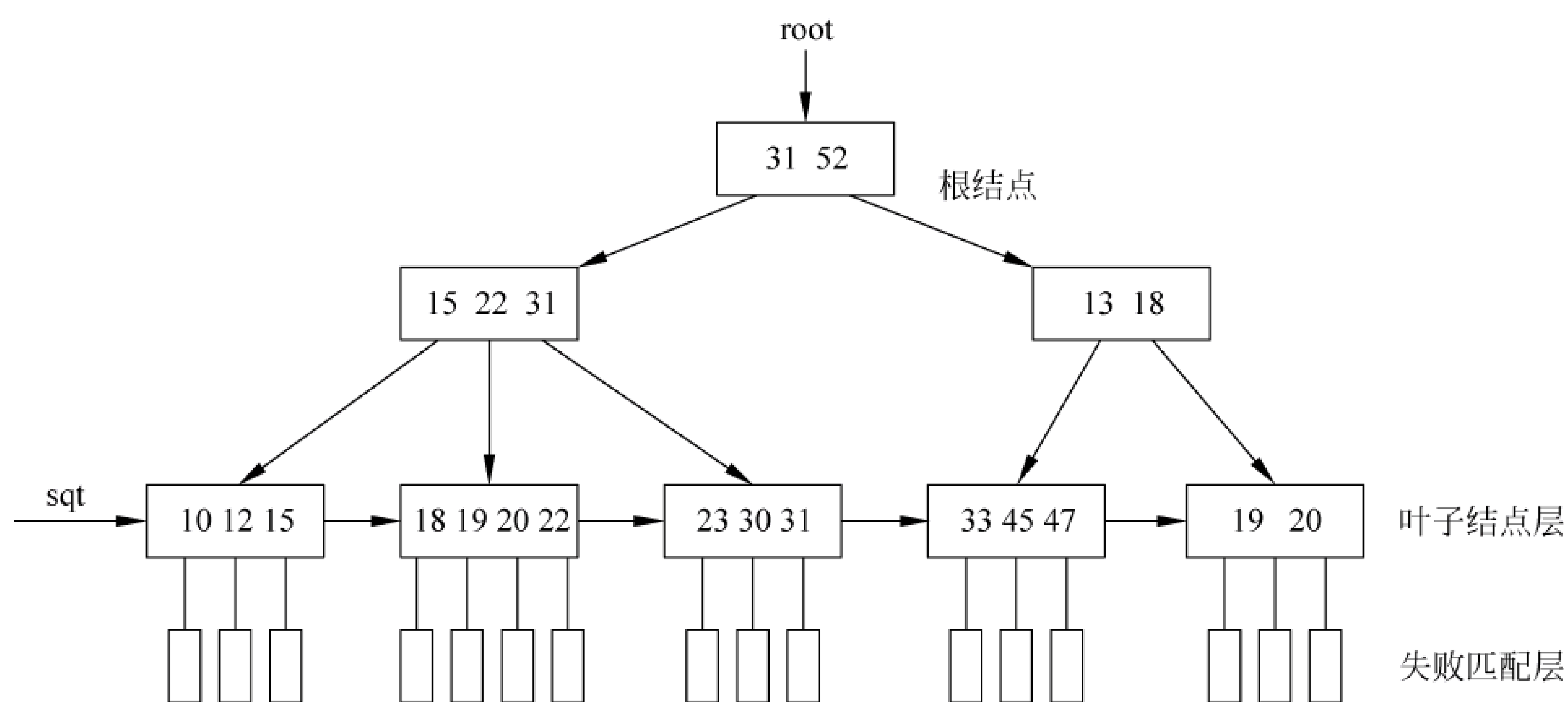


图 8.20　一棵 4 阶的 B+树

只是在分支结点上的关键字与查找值相等时，查找并不结束，要继续查到叶子结点为止，此时若查找成功，则按所给指针取出对应元素。因此，在 B+树中，不管查找成功与否，每次查找都是经过了一条从树根结点到叶子结点的路径。

2. B+树的插入

与 B—树的插入操作相似，B+树的插入也从叶子结点开始，当插入后结点中的关键字个数大于 m 时，要分裂成两个结点，它们所含关键字个数分别为$\lceil (m+1)/2 \rceil$和$\lfloor (m+1)/2 \rfloor$，同时要使它们的双亲结点中包含有这两个结点的最大关键字和指向这两个结点的指针。若双亲结点的关键字个数大于 m，则应继续分裂，以此类推。

3. B+树的删除

B+树的删除也是从叶子结点开始，当叶子结点中最大关键字被删除时，分支结点中的值可以作为「分界关键字」存在。若因删除操作而使结点中的关键字个数少于$\lceil m/2 \rceil$时，则从兄弟结点中调剂关键字或将该结点和兄弟结点合并，其过程和 B—树的删除操作相似。

8.4　哈希表查找



视频讲解

8.4.1　哈希表的基本概念

哈希表(Hash Table)又称散列表，是除顺序表存储结构、链表存储结构和索引表存储结构之外的又一种存储结构。哈希表存储的基本思路是：设要存储的对象个数为 n，设置一个长度为 m($m \geqslant n$)的连续内存单元，以静态表中每个对象的关键字 k_i($0 \leqslant i \leqslant n-1$)为自变量，通过一个称为哈希函数的函数 $h(k_i)$把 k_i 映射为内存单元的地址(或称下标)$h(k_i)$，并把该对象存储在这个内存单元中。$h(k_i)$也称为哈希地址(又称散列地址)。通常把如此构造的存储结构称为**哈希表**。

对于两个关键字 k_i 和 k_j($i \neq j$)，有 $k_i \neq k_j$($i \neq j$)，但 $h(k_i)=h(k_j)$。这种不同关键字竞争同一地址的现象叫作**哈希冲突**。通常把这种具有不同关键字，而具有相同哈希地址的诸对象称作「**同义词**」，由诸同义词引起的冲突称作**同义词冲突**。在哈希表存储结构中，同义词冲

突是很难避免的,除非关键字的变化区间小于等于哈希地址的变化区间,而当关键字取值不连续时,是非常浪费存储空间的。通常的实际情况是关键字的取值区间远大于哈希地址的变化区间。

一旦哈希表建立,在哈希表中进行查找的方法就是以要查找的关键字 k 为映射函数的自变量、以建立哈希表时使用的哈希函数 h(k)为映射函数得到一个哈希地址(设该地址中对象的关键字为 k_i),比较关键字 k 和 k_i,如果 $k=k_i$,则查找成功;否则,以建立哈希表时使用的哈希冲突函数得到新的哈希地址(设该地址中对象的关键字为 k_j),比较关键字 k 和 k_j,如果 $k=k_j$,则查找成功,否则以同样的方式继续查找,直到查找成功或查找完 m 个存储单元仍未查找到(即查找失败)为止。

8.4.2 哈希函数构造方法

构造哈希函数的目标是使得到的哈希地址尽可能均匀地分布在 n 个连续内存单元地址上,同时使计算过程尽可能简单,以达到尽可能高的时间效率。根据关键字的结构和分布的不同,可构造出许多不同的哈希函数。这里主要讨论几种常用的整数类型关键字的哈希函数构造方法。

1. 直接定址法

直接定址法是以关键字 k 本身或关键字加上某个数值常量 c 作为哈希地址的方法。直接定址法的哈希函数 h(k)为

$$h(k) = k + c$$

这种哈希函数计算简单,并且不可能有冲突发生。当关键字的分布基本连续时,可用直接定址法的哈希函数;否则,若关键字分布不连续,将造成内存单元的大量浪费。

2. 除留余数法

除留余数法是用关键字 k 除以某个不大于哈希表长度 m 的数 p,将所得的余数作为哈希地址的方法。除留余数法的哈希函数 h(k)为

$$h(k) = k \bmod p(\text{mod 为求余运算}, p \leqslant m)$$

除留余数法计算比较简单,适用范围广,是最经常使用的一种哈希函数。这种方法的关键是选好 p,使得元素集合中的每一个关键字通过该函数转换后映射到哈希表范围内的任意地址上的概率相等,从而尽可能减少发生冲突的可能性。

理论研究表明,p 取奇数就比取偶数好,**当 p 取不大于 m 的素数时,效果最好**。

3. 数字分析法

数字分析法是提取关键字中取值较均匀的数字位作为哈希地址的方法,适合于所有关键字值都已知的情况,并需要对关键字中每一位的取值分布情况进行分析。例如,有一组关键字为{92317602,92326875,92739628,92343634,92706816,92774638,92381262,92394220},通过分析可知,每个关键字从左到右的第 1,2,3 位和第 6 位取值较集中,不宜作为哈希地址,剩余的第 4,5,7 和 8 位取值较分散,可根据实际需要取其中的若干位作为哈希地址。若取最后两位作为哈希地址,则哈希地址的集合为{02,75,28,34,16,38,42,20}。

其他构造整数关键字的哈希函数的方法还有平方取中法、折叠法等。平方取中法是取关键字平方后分布均匀的几位作为哈希地址的方法;折叠法是先把关键字中的若干段作为一小组,然后把各小组折叠相加后分布均匀的几位作为哈希地址的方法。

8.4.3 哈希冲突解决方法


视频讲解

解决哈希冲突的方法有许多,可分为开放定址法和拉链法两大类。其基本思路是:当发生哈希冲突时,通过**哈希冲突函数**(设为 $h_c(k)(c=1,2,\cdots,m-1)$)产生一个新的哈希地址,使 $h_c(h_i) \neq h_c(k_j)$ 哈希冲突函数产生的哈希地址仍可能有哈希冲突问题,此时再用新的哈希冲突函数得到新的哈希地址,一直到不存在哈希冲突为止,因此有 $c=1,2,\cdots,m-1$。这样就把要存储的 n 个元素通过哈希函数映射得到的哈希地址(当哈希冲突时,通过**哈希冲突函数**映射得到的哈希地址)存储到了 m 个连续内存单元中,从而完成了哈希表的建立。在哈希表中,虽然冲突很难避免,但发生冲突的可能性却有大有小。这主要与以下 3 个因素有关。

- 与装填因子 α 有关。**填充因子**是指哈希表中已存入的元素数 n 与哈希地址空间大小 m 的比值,即 $\alpha=n/m$。α 越小,冲突的可能性越小;α 越大(最大可取 1),冲突的可能性越大。

因为 α 越小,哈希表中空闲单元的比例就越大,所以待插入元素同已插入的元素发生冲突的可能性就越小;显然,α 越小,存储空间的利用率越低。

反之,α 越大,哈希表中空闲单元的比例就越小,所以待插入元素同已插入的元素冲突的可能性就越大。显然,α 越大,存储空间的利用率越高。为了兼顾减少冲突的发生和提高存储空间的利用率这两个方面,通常将最终的 α 控制在 0.6~0.9 的范围内。

- 与采用的哈希函数有关。若哈希函数选择得当,就可使哈希地址尽可能均匀地分布在哈希地址空间上,从而减小冲突发生的可能性;否则,若哈希函数选择不当,就可能使哈希地址集中于某些区域,从而增大冲突发生的可能性。
- 与解决冲突的哈希冲突函数有关。哈希冲突函数的选择也影响发生冲突的可能性。

下面介绍几种常用的解决哈希冲突的方法。

1. 开放定址法(闭散列法)

开放定址法是一类以发生冲突的哈希地址为自变量,通过某种哈希冲突函数得到一个新的空闲的哈希地址的方法。在开放定址法中,哈希表中的空闲单元(假设其下标为 d)不仅允许哈希地址为 d 的同义词关键字使用,而且也允许发生冲突的其他关键字使用,因为这些关键字的哈希地址不为 d,所以称它们为非同义词关键字。开放定址法的名称就来自此方法的哈希表空闲单元,既向同义词关键字开放,也向发生冲突的非同义词关键字开放。至于哈希表的一个地址中存放的是同义词关键字,还是非同义词关键字,要看谁先占用它,这和构造哈希表的元素排列次序有关。

开放定址法(即以发生冲突的哈希地址为自变量,通过某种哈希冲突函数得到一个新的空闲的哈希地址的方法)有很多种,下面介绍常用的几种。

1) 线性探查法

线性探查法是从发生冲突的地址(设为 d_0)开始,依次探查 d_0 的下一个地址(当到达下标为 m−1 的哈希表表尾时,下一个探查的地址是表首地址 0),直到找到一个空闲单元为止(当 $m \geqslant n$ 时,一定能找到一个空闲单元)。线性探查法的数学递推描述公式为

$$d_0 = h(k)$$
$$d_i = (d_0 + 1) \bmod m \quad (1 \leqslant i \leqslant m-1)$$

线性探查法容易产生**堆积**问题。这是由于当连续出现若干个同义词后(设第一个同义词占用单元 d_0,这连续的若干个同义词将占用哈希表的 d_0,d_0+1,d_0+2 等单元),任何 d_0+1,d_0+2 等单元上的哈希映射都会由于前面的**同义词堆积**而产生冲突,尽管随后的这些关键字并没有同义词。

2)平方探查法

设发生冲突的地址为 d_0,则平方探查法的探查序列为:d_0+1^2,d_0-1^2,d_0+2^2,d_0-2^2,…。平方探查法的数学描述公式为

$$d_0 = h(k)$$
$$d_i = (d_0 \pm i^2) \bmod m(1 \leqslant i \leqslant m-1)$$

平方探查法是一种较好处理冲突的方法,可以避免出现堆积问题。它的缺点是不能探查到哈希表上的所有单元,但至少能探查到一半单元。

注意:求模运算的目的在于当探查范围从当前元素探查到表空间的末尾时,可以从表首空间继续,即逻辑上将一块连续空间视作一长串环形队列,使得可以遍历探查每一块空间是否可供存放同义词。

此外,开放定址法的探查方法还有伪随机序列法。解决冲突的方法还有双哈希函数法、建立一个公共溢出区等。

【例 8.11】 假设哈希表长度 m=13,采用除留余数法加线性探查法建立如下关键字集合的哈希表:{70,20,47,3,91,65,63,72,15,88,31}。

解:n=11,m=13,除留余数法的哈希函数为 h(k)=k mod p,p 应为小于或等于 13 的素数,假设 p 取值 13,当出现同义词问题时,采用线性探查法解决冲突。

建立的哈希表 ha[0..12]见表 8.1。

表 8.1 建立的哈希表 ha[0..12]

下标	0	1	2	3	4	5	6	7	8	9	10	11	12
k	91	65	14	3		70	31	20	47	72	88	63	
探查次数	1	2	2	1		1	2	1	1	3	1	1	

哈希表 ha[0..12]的创建过程见表 8.2。

表 8.2 哈希表 ha[0..12]的创建过程

数　　值	描　　述
h(70)=5	没有冲突,将 70 放在 ha[5]处
h(20)=7	没有冲突,将 20 放在 ha[7]处
h(47)=8	没有冲突,将 47 放在 ha[8]处
h(3)=3	没有冲突,将 03 放在 ha[3]处
h(91)=0	没有冲突,将 91 放在 ha[0]处
h(65)=0	有冲突
$d_0=0$, $d_1=(0+1)$ mod 13=1	**冲突已解决**,将 65 放在 ha[1]处
h(63)=11	没有冲突,将 63 放在 ha[11]处
h(72)=7	有冲突
$d_0=7$, $d_1=(7+1)$ mod 13=8	仍有冲突

数　　值	描　　述
$d_2=(8+1) \bmod 13=9$	**冲突已解决**,将 72 放在 ha[9]处
h(14)=1	有冲突
$d_0=1$, $d_1=(1+1) \bmod 13=2$	**冲突已解决**,将 14 放在 ha[2]处
h(88)=10	没有冲突,将 88 放在 ha[10]处
h(31)=5	**有冲突**
$d_0=5$,$d_1=(5+1) \bmod 13=6$	**冲突已解决**,将 31 放在 ha[6]处

2. 拉链法(开散列法)

拉链法是把所有的同义词用单链表链接起来的方法。在这种方法中,哈希表每个单元中存放的不再是元素本身,而是相应同义词单链表的头指针。由于单链表中可插入任意多个结点,所以此时装填因子 α 根据同义词的多少既可以设定为大于 1,也可以设定为小于或等于 1,通常取 α=1。

与开放定址法相比,拉链法有以下几个**优点**。

- 处理冲突简单,且无堆积现象,即非同义词之间绝不会发生冲突,因此平均查找长度较短。
- 由于各链表上的元素空间是动态申请的,故它更适合于建表前无法确定表长的情况。
- 开放定址法为减少冲突,要求装填因子 α 较小,故当数据规模较大时,会浪费很多空间,而拉链法中可取 α>1,且数据规模较大时,拉链法中增加的指针域可忽略不计,因此节省空间。
- 在用拉链法构造的哈希表中,删除元素的操作易于实现,只删去链表上相应的元素即可。

注意:与拉链法的优点相比,开放地址法构造的哈希表,删除元素不能简单地将被删元素的空间置为空,否则将截断在它之后填入哈希表的同义词元素的查找路径,这是因为在开放地址法中,空地址单元(即开放地址)是查找失败的条件。因此,在用开放地址解决冲突构造的哈希表上执行删除操作,只能在被删元素上做删除标记,而不能真正删除元素。只有当运行到一定阶段经过整理后,才能真正删除有标记的元素。

拉链法也有**缺点**:指针需要额外的空间,故当元素规模较小时,还是开放定址法较节省空间,而且若将节省的指针空间用来扩大哈希表的规模,可使装填因子变小,这又减少了开放定址法中的冲突,从而提高了平均查找速度。

【例 8.12】 假设哈希表长度 m=13,采用除留余数法加拉链法建立如下关键字集合的哈希表:(70, 20, 47, 3, 91, 65, 63, 72, 14, 88, 31)。

解:n=11,m=13,除留余数法的哈希函数为 h(k)=k mod p,p 应为不大于 m 的素数,假设 p 取值 13。

当出现同义词问题时,采用拉链法解决冲突,则有

h(70)=5,h(20)=7,h(47)=8,h(3)=3,

h(91)=0,h(65)=0,h(63)=11,h(72)=7,

h(14)=1,h(88)=10,h(31)=5。

采用拉链法解决冲突建立的链表如图 8.21 所示。

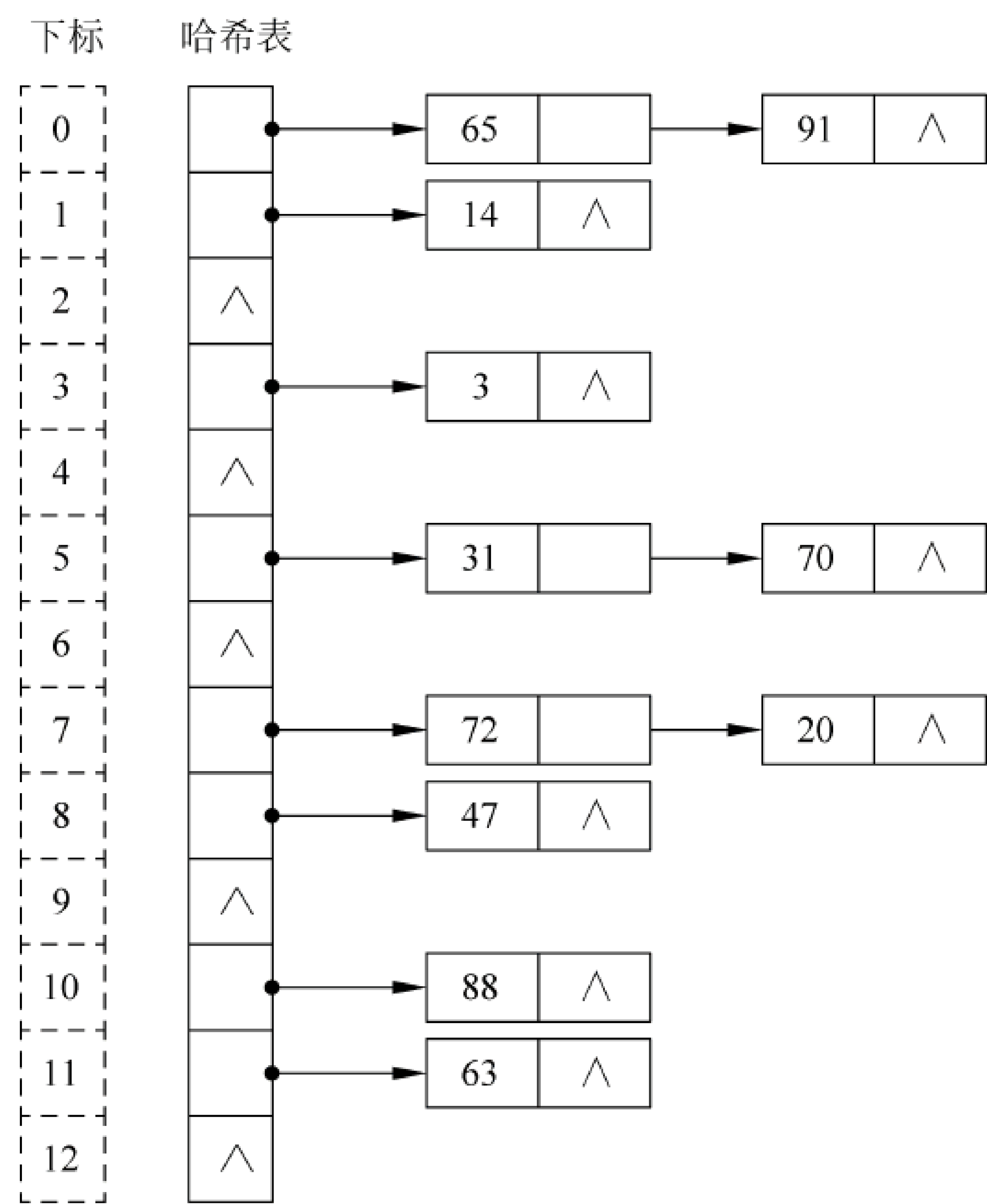


图 8.21　采用拉链法解决冲突建立的链表

8.4.4　哈希表上的查找分析

哈希表上的运算有查找、插入和删除，其中主要是查找，这是因为哈希表主要是用于快速查找，且插入和删除均要用到查找操作，故本节仅给出建立哈希表、删除表中元素、查找表中元素的相关操作，并着重分析在哈希表上进行查找运算的性能。

下面给出有关的类型说明。

```
#define MaxSize 100                     //定义最大哈希表长度
#define NULLKEY -1                      //定义空关键字值
#define DELKEY -2                       //定义被删关键字值
typedef int KeyType;                    //关键字类型
typedef char * InfoType;                //其他数据类型
typedef struct {
    KeyType key;                        //关键字域
    InfoType data;                      //其他数据域
    int count;                          //探查次数域
} HashNode;
typedef HashNode HashTable[MaxSize];    //哈希表类型
```

1. 查找

哈希表的查找过程和建表过程相似。假设给定的值为 k，根据建表时设定的散列函数 h 计算出哈希地址 h(k)，若表中该地址单元不为空(关键字值 NULLKEY 表示为空)且该地址的关键字不等于 k，则按建表时设定的处理冲突的方法找下一个地址(这里采用线性探查

法找下一个地址),如此反复下去,直到某个地址单元为空(查找失败,返回-1)或者关键字比较相等(查找成功,返回该地址)为止。对应的算法如下。

```
int SearchHT(HashTable ha,int p,KeyType k)          //在哈希表中查找关键字 k
{
int i=0,adr;
adr=k % p;
while (ha[adr].key!=NULLKEY && ha[adr].key!=k)
{
  i++;                                              //采用线性探查法找下一个地址
  adr=(adr+1) % p;
}
if (ha[adr].key==k)                                 //查找成功
  return adr;
else                                                //查找失败
  return -1;
}
```

2. 删除

在采用开放定址法处理冲突的哈希表上执行删除操作,只能在被删元素上做删除标记,而不能真正删元素(参见 8.4.3 节中开放定址法与拉链法比较的讨论)。这里,设置标记 DELKEY,以示区分。对应的算法如下。

```
int DeleteHT(HashTable ha,int p,int k,int &n)
{                    //删除哈希表中的关键字 k
 int adrr = SearchHT(ha, p, k);
 if (adrr != -1)
 {                   //在哈希表中找到该关键字
  ha[adrr].key = DELKEY;
  n--;               //哈希表长度减 1
  return 1;
 }
 else
  return 0;          //在哈希表中未找到该关键字
}
```

3. 插入和建表

建表时,首先将表中各结点的关键字清空,使其地址重新开放,然后调用插入算法将给定的关键字序列依次插入表中。插入算法首先调用查找算法,若在表中找到了待插入的关键字,则插入失败;若在表中找到一个开放地址,则将待插入的结点插入其中,即插入成功。对应的算法如下。

```
void InsertHT(HashTable ha,int &n,KeyType k,int p)
{ //将关键字 k 插入到哈希表中
 int i,adr;
```

```
adr=k % p;
 if (ha[adr].key==NULLKEY || ha[adr].key==DELKEY)
 {
  //x[j]可以直接放在哈希表中
  ha[adr].key=k;
  ha[adr].count=1;
 }
 else
{                          //发生冲突时采用线性探查法解决冲突
  i=1;                     //i 记录 x[j]发生冲突的次数
  do {
     adr=(adr+1) % p;
     i++;
   } while (ha[adr].key != NULLKEY && ha[adr].key != DELKEY);
 ha[adr].key=k;
 ha[adr].count=i;
}
 n++;
}
void CreateHT(HashTable ha,KeyType x[],int n,int m,int p)
{ //创建哈希表
 int i,n1=0;
 for (i=0;i<m;i++)
 { //哈希表置初值
     ha[i].key=NULLKEY;
     ha[i].count=0;
 }
 for (i=0;i<n;i++)
     InsertHT(ha,n1,x[i],p);
}
```

4. 性能分析

插入和删除的时间均取决于查找,故只分析查找操作的时间性能。查找成功时的平均查找长度是查找到表中已有表项的平均探查次数,是找到表中各个已有表项的探查次数的平均值。查找不成功的平均查找长度是在表中查找不到待查的表项,但找到插入位置的平均探查次数,是在表中所有可能的散列的位置上插入新元素时为找到空位置的探查次数的平均值。例如,在各结点查找概率相等的情况下,在例 8.11 和例 8.12 构建的哈希表中查找成功时的平均查找长度如下。

$$ASL_{成功}=\frac{1\times7+2\times3+3\times1}{11}=1.4545(线性探查法)$$

$$ASL_{成功}=\frac{1\times8+2\times3}{11}=1.2727(拉链法)$$

- 式中,$\frac{1}{11}$表示 11 个结点中每个结点查找成功的概率相等。
- **线性探查法**中,1×7,1×3 和 3×1 分别表示探查 1,2 和 3 次的结点各有 7,3 和 1 个。这里,探查次数即和待查关键字 k 的比较次数(参见表 8.1)。
- 在**拉链法**中,1×8 和 2×3 分别表示比较 1 次和 2 次的结点各有 8,3 个(参见图 8.21)。

下面仍以例 8.11 和例 8.12 的哈希表为例，分析在等概率情况下，查找**不成功**时平方探查法和拉链法的平均查找长度。

在表 8.1 所示的线性探查法中，假设待查关键字 k 不在该表中：

- 若 h(k)＝m，m＝＝NULL，探查可以结束，即比较次数为 1。
- 若 m!＝NULL，必须在 k 和 ha[m]中的关键字进行比较（不等）之后，再与同义词 ha[m＋1]进行比较，此时若 ha[m＋1]＝NULL，则探查可以结束，即比较次数为 2。
- ……
- 若 ha[m＋i]!＝NULL，因为逆向解释建表时线性探查法解决冲突的方法为：设发生冲突的地址为 d_0，d_0 之后的每一个空间单元都有可能存放有要探查的元素 k，需重复这个过程，直到遇到第一个 NULL 才可得出查找失败的结论。

线性探查法解决冲突的弊端：使下一同义词 d_i 下标 i 向后增一，并关于空间长度进行求模，直到遇到第一个空闲的哈希空间地址，这个过程可能造成同义词堆积于 d_0 之后无缝紧跟的连续空间单元上。同时，同义词堆积越多，越不利于插入和查找，致使平均查找长度值直线上升。

由此可得查找不成功时的平均查找长度为

$$ASL_{失败}=\frac{5+4+3+2+1+8+7+6+5+4+3+2+1}{13}\approx 3.9231（线性探查法）$$

在图 8.21 所示的拉链法中，若待查关键字 k 的哈希地址为 d＝h(k)，且第 d 个链表上具有 i 个结点，则当 k 不在此表上时，就需做 i 次关键字的比较（不包括空指针判定），因此查找不成功时的平均查找长度为

$$ASL_{失败}=\frac{2+1+0+1+0+2+0+2+1+0+1+1+0}{13}\approx 0.8462（线性探查法）$$

从上述例子可以看出，由同一个哈希函数、不同的解决冲突方法构造的哈希表，其平均查找长度是不相同的。

【例 8.13】 用关键字序列{3，18，29，14，17，11}构造一个哈希表，哈希表的存储空间是一个下标从 0 开始的一维数组，哈希函数为 H(key)＝(key×3) MOD 7，处理冲突采用线性探测法，要求装填(载)因子为 0.75。

(1) 画出构造的哈希表。

(2) 分别计算等概率情况下，查找**成功**和查找**不成功**的平均查找长度。

解：

(1) n＝6，α＝0.7＝n/m，则 m＝n/0.75＝8。

计算各关键字存储地址的过程见表 8.3。

表 8.3 计算各关键字存储地址的过程

数　值	描　述
h(3)＝(3×3) MOD 7＝2	
h(18)＝(18×3)MOD 7＝5	
h(29)＝(29×3)MOD 7＝3	
h(14)＝(14×3)MOD 7＝0	
h(17)＝(17×3)MOD 7＝2	冲突

续表

数　　值		描　　述
	$d_1=(2+1)\text{MOD } 8=3$	仍冲突
	$d_2=(3+1)\text{MOD } 8=4$	
$h(11)=11\times 3\text{ MOD } 7=5$		冲突
	$d_1=(5+1)\text{MOD } 8=6$	

构造的哈希表见表 8.4。

表 8.4　构造的哈希表

下标	0	1	2	3	4	5	6	7
关键字	14		3	29	17	18	11	
探测次数	1		1	1	3	1	2	

注意：由于任一关键字 k，表中无本关键字的同义词时，$h(k)=(k\times 3)\text{ MOD } 7$ 的值只能是 0～6，而解决冲突时探查的地址 $d_i=(d_{i-1}+i)\text{ MOD } 8(1\leqslant i\leqslant 8-1)$的取值为 0～7，需区分两者。

(2) 在等概率情况下：

$$ASL_{成功}=\frac{1\times 4+2\times 1+3\times 1}{6}=1.5$$

在不成功的情况下，探测次数见表 8.5。

表 8.5　不成功时探测次数

下标	0	1	2	3	4	5	6	7
关键字	14		3	29	17	18	11	
探测次数	2	1	6	5	4	3	2	1

所以有

$$ASL_{失败}=\frac{2+1+6+5+4+3+2+1}{6}=4$$

一般情况下，假设哈希函数是均匀的，则可以证明：不同的解决冲突方法得到的哈希表的平均查找长度不同。表 8.6 列出了用几种不同的方法解决冲突时哈希表的平均查找长度。从表 8.6 中看到，哈希表的平均查找长度不是元素个数 n 的函数，而是装填因子 α 的函数。因此，在设计哈希表时，可选择 α 控制哈希表的平均查找长度。

表 8.6　用几种不同的方法解决冲突时哈希表的平均查找长度

解决冲突的方法	ASL	
	成功的查找	不成功的查找
线性探查法	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
平方探查法	$-\frac{1}{\alpha}\ln(1-\alpha)$	$\frac{1}{1-\alpha}$
拉链法	$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}\approx\alpha$

8.5 STL 中的查找

排序查找是 STL 中常用的算法。特定的算法总搭配特定的数据结构。例如，经典的红黑树、散列表等都是为了解决查找问题发展起来的。STL 算法库中提供的查找相关算法主要有两类：在范围中找元素(find，find_if，find_if_not，find_first_of)；在范围中找范围(find_end，search，search_n)。

1. 单个元素查询

```
find()比较条件为相等的查找，从给定区间中查找单个元素，定义：
template <class InputIteraor,class T>
InputIteraor find(InputIterator first,InputIterator last,const T&val);
```

在[first，last)范围中查找第一个等于 val 的值，如果找到，则返回该值的迭代器，否则返回迭代器 last。

例如，在 myvector 中查找 20。

```
int myints[]={10,20,30,40};
std::vector<int> myvector(myints,myints+4);
it=find(myvector.begin(),myvector.end(),20);
if(it!=myvector.end())
  std::cout<<"Element found in myvector:"<< *it<<'\n';
else
  std::cout<<"Element not found in myvector:"<< '\n';
```

find_if()自定义比较函数，从给定区间中找出满足比较函数的第一个元素。例如，从 myvector 中查找能被 5 整除的第一个元素。

```
bool cmpFunction(int i)
{
  return ((i%5)==0);
}
 it=std::find_if(myvector.begin(),myvector.end().cmpFunction);
 std::cout<<"first:"<< *it<<std::endl;
```

2. 区间查找

search()查找子区间首次出现的位置。例如，从 myvector 中查找出现子区间[20，30)的位置。

```
int needle1[]={20,30};
it=std::search(myvector.begin(),myvector.end(),needle1,needle1+2);
if(it!=myvector.end())
  std::cout<<"needle1 found at position"<<(it-myvector.begin())<<'\n';
```

search 支持自定义比较函数。例如,查询给定区间中每个元素比目标区间小 1 的子区间。

```
bool cmpFunction(int i,int j)
{
 return(i-j==1);
 }
int myints[]={1,2,3,4,5,1,2,3,4,5};
std::vector<int> haystack(myints,myints+10);
int needle2[]={1,2,3};
it=std::search(haystack.begin(),haystack.end(),needle2,needle2+3,cmpFunction);
```

8.6 综合案例——拼写检查问题

1. 问题描述

微软的 Word 有一个拼写检查功能,如果拼错单词,它会用红线标出,以示提醒,然后给出可能正确的单词。编程实现类似的系统:给定一个词表以及一个待检查的单词,判断这个单词是否在词表中,如果不在词表中,程序应该给出一个相似的单词。

在寻找相似的单词时,只需要考虑如下几个简单的情况。

- 漏写了一个字母,如把 abacus 误拼写为 abacs。
- 多写了一个字母,如把 abacus 误拼写为 abaacus。
- 将某处的一个字母写成了另一个字母,如 abacus 误拼写为 abacup。

输入格式:输入数据的第一行是一个由小写字母组成的字符串,表示要进行拼写检查的单词,第二行是一个数 N(1≤N≤100),表示词表中词的数目,接下来有 N 行,每行都是一个由小写字母组成的字符串,代表词表中的每一个单词。所有字符串的长度都为 2~20。

输出格式:仅输出一个字符串。

- 如果要检查的单词在词表中出现,则原样输出该单词。
- 如果要检查的单词在词表中未出现,但在词表中找到相似的单词,则输出在词表中和它相似的那个单词。如果在词表中找到多个相似单词,仅输出在输入文件中最靠前的一个。
- 如果要检查的单词在词表中未出现,并且在词表中找不到与它相似的单词,则输出 NOANSWER。

样例输入:

```
abstaine
4
abacus
abstract
abstain
abstainer
```

样例输出：

```
abstain
```

2. 解题思路

- 如果用户输入的待检测单词和词典中的已有单词相同，则直接输出匹配成功。
- 主串和词典串长度相等，比较不同的字符数，但最多只允许有且仅有一个不同。
- 主串和词典串长度相差的绝对值为 1。

  长度少 1，检测到不同时让 temp 多走一位，然后扫描时必须完全相等。

  长度多 1，检测到不同时让 temp 多走一位，然后扫描时必须完全相等。

3. 代码实现

```
#include <cstdio>
#include <cstring>
#define max_len 21
int main() {
    char temp[max_len], answer[max_len], A[max_len];
    scanf("%s", temp);
    int n, i, j, check, flag = 0, len, temp_len;
    temp_len = strlen(temp);
    scanf("%d", &n);
    for (i = 0; i < n; ++i) {
        scanf("%s", A);
        if (!strcmp(temp, A)) {
            printf("%s", temp);
            return 0;
    }//若相等，则直接输出结果跳出程序即可
    len = strlen(A);
    if (!flag) {
        //如果之前检测到有相似单词，就不用再检测了，
        //且只需考虑 len 与 temp_len 相差不超过 1 的情形
        if (len == temp_len) {
            //长度相等，比较不同的字符数，必须有且仅有一个不同
            //此时 check 为 0，!check 为真，否则 check 为负数，!check 为假
            for (check = 1, j = 0; j < len; ++j)
                if (A[j] != temp[j]) --check;
            if (!check) {
                strcpy(answer, A);
                flag = 1;
            }
        } else if (len == temp_len - 1) {
            //长度少 1，检测到不同时让 temp 多走一位，然后扫描时必须完全相等
            j = 0; check = 1;
            while (A[j] == temp[j])
                ++j;
            while (j < len) {
```

```
            if(A[j] != temp[j + 1]) {
                check = 0;
                break;
            }
            ++j;
        }
        if (check) {
            strcpy(answer, A);
            flag = 1;
        }
    } else if (len == temp_len + 1) {
        //长度多1,检测到不同时让 temp 多走一位,然后扫描时必须完全相等
        j = 0; check = 1;
        while (A[j] == temp[j])
            ++j;
        while (j < len) {
            if(A[j + 1] != temp[j]) {
                check = 0;
                break;
            }
            ++j;
        }
        if (check) {
            strcpy(answer, A);
            flag = 1;
        }
    }
  }
 }
 if (!flag) printf("NOANSWER");
 else printf("%s", answer);
 return 0;
}
```

本章小结

本章主要介绍了查找运算的基本知识,主要学习要点如下。

- 理解查找的基本概念,包括静态查找表和动态查找表、内查找和外查找之间的差异以及平均查找长度等。
- 理解并掌握静态表的各种查找算法,包括顺序查找、折半查找和分块查找的基本思路、算法实现和查找效率等。
- 理解并掌握各种树表的查找算法,包括二叉排序树、AVL 树、B－树和 B＋树的基本思路、算法实现和查找效率等。
- 掌握哈希表查找技术以及哈希表与其他存储方法的本质区别。
- 做到灵活运用各种查找算法解决一些综合应用问题。

第9章　排　序

9.1　排序的基本概念

排序(sorting)又称分类,是日常工作和软件设计中常用的运算之一。在实际应用中,为了提高查询速度,需要将无序序列按照某个关键字值(关键字)递增或递减组织成有序序列。排序的主要目的是实现快速查找,日常生活中通过排序后进行检索的例子屡见不鲜,如电话簿、病例、档案室中的档案、图书馆和各种词典的目录表等,都需要对有序数据进行操作。由于需要排序的数据表的基本特性可能存在差异,因而产生了许多不同的排序方法。因此,如何合理地组织数据的逻辑顺序,获得具有最佳时间、空间效率的排序方法是本章要讨论的主题。

为了便于讨论,首先对排序进行定义。

假设含 n 个记录的序列为

$$\{R_1, R_2, \cdots, R_n\} \tag{9-1}$$

其对应的关键字序列为

$$\{K_1, K_2, \cdots, K_n\} \tag{9-2}$$

须确定 1,2,…,n 的一种排列$\{P_1, P_2, \cdots, P_n\}$,使其相应的关键字 K_{pi}满足如下的非递减(非递增同理)关系,即满足:

$$K_{p1} \leqslant K_{p2} \leqslant \cdots \leqslant K_{pn}$$

即使式(9-1)的序列称为一个关键字有序序列:

$$\{R_{p1}, R_{p2}, \cdots, R_{pn}\} \tag{9-3}$$

这样的操作称为**排序**。

排序可以进行多种分类。简单的分类如递增排序和递减排序:如果排序的结果是按关键字从小到大的次序排列的,如式(9-3),就是递增排序,否则就是递减排序。排序也可分为稳定排序和不稳定排序:假设 $K_i=K_j(1\leqslant i\leqslant n, 1\leqslant j\leqslant n, i\neq j)$,且在排序前的序列中 R_i 领先于 R_j(若 $i<j$)。若在排序后的排序中 R_i 仍领先于 R_j,即那些记录经过排序后相对次序仍然保持不变,则称这种排序方法是**稳定**的;反之,若 R_j 领先于 R_i,则称所用的方法是**不稳定**的。

排序方法还可分为内部排序和外部排序。在排序中,若数据表中的所有记录的排列过程都在内存中进行,则称为**内部排序**。由于待排序的记录数量太多,在排序过程中不能同时把全部记录放在内存,需要不断地通过在内存和外存之间交换数据元素完成整个排序的过程,称为**外部排序**。在外部排序情况下,只有部分记录进入内存,在内存中进行内部排序,待排序完成后再交换到外部存储器中加以保存,然后再将其他待排序的记录调入内存继续排序。

这一过程需要反复进行,直到全部记录排出次序为止。显然,内部排序是外部排序的基础。

和许多算法一样,对各种排序算法性能的评价主要从两个方面考虑:一是时间复杂度;二是空间复杂度。排序算法的时间复杂度可用排序过程中记录之间关键字的比较次数与记录的移动次数衡量。在本章各节中讨论算法的时间复杂度时,一般都按平均时间复杂度进行估算;对于那些受数据表中记录的初始排列和记录数目影响较大的算法,按最好情况和最坏情况分别进行估算。排序算法的空间复杂度是指算法在执行时所需的附加存储空间,也就是额外需要的用来临时存储数据的内存使用情况。若无特别说明,本章均假定排序的记录序列采用顺序表结构存储,即数组存储方式,并假定是按关键字递增方式排序。设待排序的顺序表中数据元素的类型定义如下。

```
typedef struct {
 KeyType key;                //关键字 key
elemtype data;               //其他数据项 data
}RecordType;                 //待排序元素类型
```

9.2 插入排序

当序列中的记录个数较少时,插入排序是一种有效的算法。插入排序的原理类似于在扑克游戏中的排序过程。开始时,所有扑克牌面朝下放在桌子上,我们左手为空。然后,每次从桌子上拿走一张牌并将它插入到左手中正确的位置。为了找到一张牌的正确位置,从右到左将它与已经在手中的每张牌进行比较,直至找到其正确位置并插入,如图 9.1 所示。桌子上的牌堆从顶至底依次拿在左手中,而左手中的牌总是排序好的,归纳可知最后所得即为正确排序。



图 9.1 插入排序思想

插入排序是一种重要的排序方法,存在许多变形,如直接插入排序、二叉插入排序、希尔排序、表插入排序、多表插入排序等。这里重点讨论直接插入排序和希尔排序。

9.2.1 直接插入排序



视频讲解

直接插入是最简单,也是直接的方法。其基本原理是:顺次从无序表中取出记录 $R_i(1\leqslant i\leqslant n)$,与有序表中记录的关键字逐个进行比较,找出其应该插入的位置,再将此位置及其之后的所有记录依次向后顺移一个位置,将记录 R_i 插入其中。

具体地说,假设待排序的 n 个记录为$\{R_1,R_2,\cdots,R_n\}$,初始有序表为$[R_1]$,无序表为$[R_2,\cdots,R_n]$。当插入第 i 个记录 $R_i(2\leqslant i\leqslant n)$时,有序表为$[R_1,\cdots,R_{i-1}]$,无序表为$[R_i,\cdots,R_n]$。将关键字 K_i 依次与 $K_{i-1},K_{i-2},\cdots,K_1$ 进行比较,找出其应该插入的位置,将该位置及其以后的记录向后顺移,插入 R_i,即完成序列中第 i 个记录的插入。当完成序列中第 n 个记录 R_n 的插入后,整个序列排序完毕。

因此,向有序表中插入记录,主要完成如下操作。

step1：为待排序的记录查找合适插入位置。

step2：移动插入点及其以后的记录，空出插入位置。

step3：插入记录。

显然，直接插入排序的每趟排序只处理一个记录，即每趟排序将一个待排序的记录插入到前面有序的子表中，直至最后一个记录插入完成。向有序表插入记录时，首先查找其在有序表中合适的插入位置，因为是在有序表中查找，所以既可以采用顺序查找，也可以采用折半查找方法搜索插入位置，在此采用顺序查找的方式进行，且待排序记录和有序表中的记录从后向前依次进行比较，这样可方便地实现边比较边将较大记录向后移动的效果。同样，也可以采用折半查找的方式进行插入位置的搜索，此刻的插入排序方法被称为二叉插入排序。关于折半查找过程及二叉插入排序过程，在此不再赘述，读者可自行学习。

以序列 A={5,2,4,6,1,3}为例，插入排序过程如图 9.2 所示。下标 j 指出正被排序的当前元素。在 for 循环(循环变量为 j)的每次迭代的开始，包含元素 A[0,…,j-2]的字数组构成了当前排序好的有序序列。每次迭代中，黑色的长方形保存取自 A[j]的关键字，并与左边的有序序列中的值依次进行比较。向右的箭头指出数组在比较后向右移动一个位置，向左的箭头指出当前元素的关键字被移动到正确的位置。图 9.2(a)～(e)为 for 循环的迭代，图 9.2(f)为最终排序好的数组。

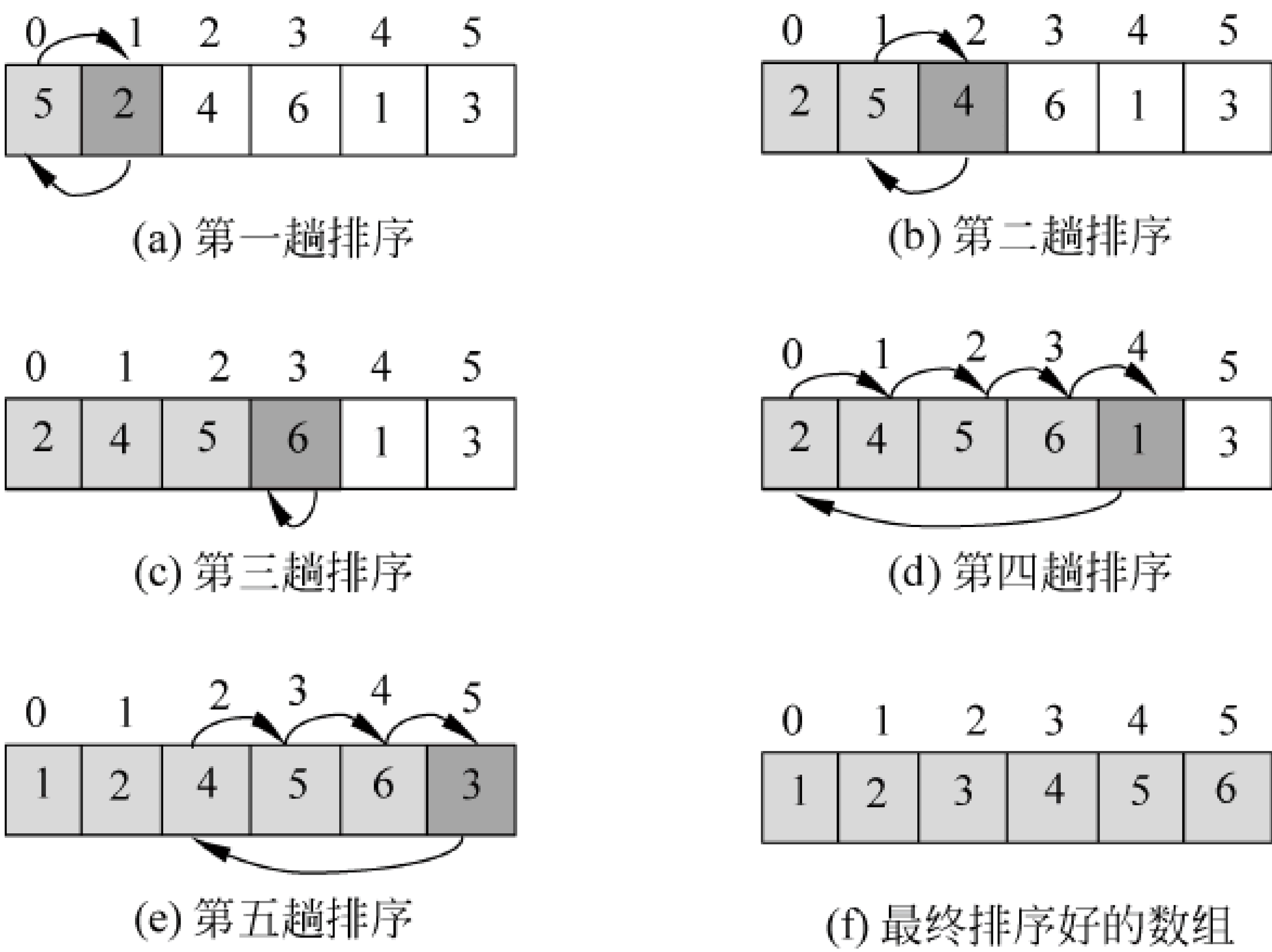


图 9.2　插入排序过程

对于直接插入排序算法实现，假设待排序的序列为数组 A[0,1,2,…,n-1]，包含长度为 n 的要排序的一个序列。排序算法实现如下。

```
void Insert_Sort(RecordType A[ ], int n)
{int i,j;
 RecordType temp;      //设置缓冲区
 for(j=1,j<=n-1;j++)
 {
  temp=A[j];
  i=j-1;
  while(i>=0&&A[i].key>temp.key)
```

```
        { A[i+1]=A[i]; i--;}
          A[i+1]=temp;
      }
   }
```

直接插入排序是稳定的,因为它在搜索插入位置时遇到关键字值相等的记录时就停止操作,不会把关键字值相等的两个数据交换位置。并且,算法在数组 A 中重排这些数,在任何时候,最多只有其中的常数个数字存储在数组外面。该算法仅需要一个记录的辅助存储空间 temp,空间复杂度为 O(1),我们称这样的算法为"原址排序"。

正如我们之前所分析的,整个算法执行 for 循环 n－1 次,每次循环中的基本操作是比较和移动,其总次数取决于数据表的初始特性,可能有以下几种情况。

(1) 当初始记录序列的关键字基本是递增排列时,这是最好的情况。算法中,while 语句的循环体执行次数为 0,因此,在一趟排序中关键字的比较次数为 1,即 temp 与 A[i]的关键字比较。而移动次数为 2,即 A[j]移动到 temp 中,temp 移动到 A[i+1]中。所以,整个排序过程中的比较次数和移动次数分别为(n－1)和 2×(n－1),此时其时间复杂度为 O(n)。

$$C_{min} = \sum_{j=1}^{n-1} 1 = n - 1$$

$$M_{min} = \sum_{j=1}^{n-1} 2 = 2(n - 1)$$

(2) 当初始数据序列的关键字序列基本是递减排列时,这是最坏的情况。在第 j 趟排序时,while 语句内的循环体执行次数为 j。因此,关键字的比较次数为 j,而移动次数为 j+2。所以,整个排序过程中的比较次数和移动次数分别为

$$C_{max} = \sum_{j=1}^{n-1} j = \frac{(n-1)(n+2)}{2}$$

$$M_{max} = \sum_{j=1}^{n-1} (j+2) = \frac{(n-1)(n+4)}{2}$$

此时其时间复杂度为 $O(n^2)$。

一般情况下,可认为出现各种排列的概率相同,因此取上述两种情况的平均值作为直接插入排序关键字的比较次数和记录移动次数,约为 $n^2/4$,所以其时间复杂度为 $O(n^2)$。

根据上述分析得知,当原始序列越接近有序时,该算法的执行效率越高。


视频讲解

9.2.2 希尔排序

希尔排序(Shell's Sort)又称缩小增量排序(Diminishing Increment Sort)。它是希尔(D. L. Shell)于 1959 年提出的插入排序的改进算法。如前所述,直接插入排序算法的时间性能取决于数据的初始特性,一般情况下,它的时间复杂度为 $O(n^2)$。但是,当待排序列为正序或基本有序时,时间复杂度则为 O(n)。因此,若能在一次排序前将排序序列调整为基本有序,则排序的效率就会大大提高。正是基于这样的考虑,希尔提出了改进的插入排序方法。

希尔排序的基本思想是:先将整个待排记录序列分割成若干小组(子序列),分别在组内进行直接插入排序,待整个序列中的记录"基本有序"时,再对全体记录进行一次直接插入

排序。希尔排序的具体步骤如下。

step1：首先取一个整数 $d_1<n$，称之为增量，将待排序的记录分成 d_1 个组，凡是距离为 d_1 倍数的记录都放在同一个组，在各组内进行直接插入排序，这样的一次分组和排序过程称为一趟希尔排序。

step2：再设置另一个新的增量 $d_2<d_1$，采用与上述相同的方法继续进行分组和排序过程。

step3：继续取 $d_{i+1}<d_i$，重复步骤 **step2**，直到增量 $d=1$，即所有记录都放在同一个组中。

注意：通常增量序列 d_i 的取值规则为 $d_1=\lfloor n/2\rfloor$，$d_2=\lfloor d_1/2\rfloor$，……直至 $d=1$ 为止。

以关键字分别为 58，46，72，95，84，25，37，58，63，12 的无序序列为例，用希尔排序法进行排序。图 9.3 给出了希尔排序的整个过程，用同一连线上的关键字表示其所属的记录在同一组。为区别具有相同关键字 58 的不同记录，用下画线标记后一个记录的关键字。第一趟排序时，取 $d_1=\lfloor 10/2\rfloor=5$，凡是间隔为 5 的记录被划分在一个组内，于是整个序列被划分成 5 组，分别为{58，25}，{46，37}，{72，58}，{95，63}，{84，12}，对各组内的记录进行直接插入排序，得到第一趟排序结果，如图 9.3(a)所示。



图 9.3 希尔排序过程

第二趟排序时，取 $d_2=\lfloor d_1/2\rfloor=2$，将第一趟排序的结果分成 2 组，分别为{25，<u>58</u>，12，46，95}，{37，63，58，72，84}。再对各组内的记录进行直接插入排序，得到第二趟排序结果，如图 9.3(b)所示。

第三趟排序时，取 $d_3=\lfloor d_2/2\rfloor=1$，所有的数据记录分成 1 组{12，37，25，58，46，63，<u>58</u>，72，95，84}，此时序列基本“有序”，对其进行直接插入排序，最后得到希尔排序的结果，如图 9.3(c)所示。

希尔排序算法实现如下。

```
void ShellSort(RecordType  A[ ], int n)
{
    int i, j, d;
    RecordType temp;
    d = n / 2;
    while (d > 0) {
            for (i = d; i <n; i++) {
                    j = i - d;
                    while(j>=0)
                            if (A[j].key >A[j + d].key) {
                                    temp = A[j];
                                    A[j] = A[j + d];
                                    A[j + d] = temp;
                                    j = j - d;
                            }
                            else j = -1;
            }
            d = d / 2;
    }
}
```

由上述代码可知,算法中约定初始增量 d 为已知;算法中采用简单的取增量值的方法,从第二次起取增量值为其前次增量值的一半。在实际应用中,可能有多种取增量的方法,并且不同的取值方法对算法的时间性能有一定的影响,因而,一种好的取增量的方法是改进希尔排序算法时间性能的关键。并且,希尔排序会使关键字相同的记录被分在不同的组中,交换相对位置,所以希尔排序是**不稳定**的。

通常,希尔排序开始时增量较大,分组较多,每组的记录数较少,故各组内直接插入过程较快。随着每一趟中增量 d_i 逐渐缩小,分组数逐渐减少,虽然各组的记录数目逐渐增多,但由于已经按 d_{i-1} 作为增量排过序,使序列表较接近有序状态,所以新的一趟排序过程也较快。因此,希尔排序在效率上较直接插入排序有较大的改进。希尔排序的时间复杂度约为 $O(n^{1.3})$,它实际所需的时间取决于各次排序时增量的取值。大量研究证明,若增量序列取值较合理,希尔排序时关键字比较次数和记录移动次数在 $n^{1.25}\sim1.6n^{1.25}$。这是在利用直接插入排序作为子序列排序方法的情况下得到的。

9.3 交换排序

利用交换记录的位置进行排序的方法称为**交换排序**。其基本思想是:两两比较待排序记录的关键字,如果逆序,就进行交换,直到所有记录都排好序为止。常用的交换排序方法主要有冒泡排序和快速排序。快速排序是一种分区交换排序法,是对冒泡排序方法的改进。

9.3.1 冒泡排序

冒泡排序(Bubble Sort)的算法思想是：设待排序序列有 n 个记录，首先将第一个记录的关键字 R_1.key 和第二个记录的关键字 R_2.key 进行比较，若 R_1.key>R_2.key，就交换记录 R_1 和 R_2 在序列中的位置；然后继续对 R_2.key 和 R_3.key 进行比较，并作相同的处理；重复此过程，直到关键字 R_{n-1}.key 和 R_n.key 比较完成。其结果是 n 个记录中关键字最大的记录被交换到序列的最后一个记录的位置上，即具有最大关键字的记录被“下沉”到最后，这个过程称为一趟冒泡排序。然后进行第二趟冒泡排序，对序列中的前 n−1 个记录进行同样的操作，使序列中关键字次大的记录被交换到序列的第 n−1 位置上；第 i 趟冒泡排序是从 R_1 到 R_{n-i+1} 依次比较相邻两个记录的关键字，并在“逆序”时交换相邻记录，其结果是这 n−i+1 个记录中关键字最大的记录被交换到 n−i+1 位置上。每一趟排序都有一个相对大的数据被交换到后面，就像一块块“大”石头不断往下沉，最大的总是最早沉下；而具有较小关键字的记录则不断向上(前)移动位置，就像水中的气泡逐渐向上飘浮一样，冒到最上面的是关键字值最小的记录。这种排序方法称为冒泡排序。

对有 n 个记录的序列最多做 n−1 趟冒泡，就会把所有记录依关键字大小排好序。如果在某一趟排序中都没有发生相邻记录的交换，表示在该趟之前已达到排序的目的，整个排序过程可以结束。在操作实现时，常用一个标志位 flag 标示在第 i 趟是否发生了交换，若在第 i 趟发生过交换，则置 flag=false(或 0)；若第 i 趟没有发生交换，则置 flag=true(或 1)，表示在第 i−1 趟已经达到排序目的，可结束整个排序过程。

假设有 9 个记录，关键字分别为{6,5,3,1,8,7,2,4,$\underline{5}$}，用冒泡排序方法排序。冒泡排序过程如图 9.4 所示。



图 9.4 冒泡排序过程

执行六趟冒泡排序后，就完成了整个排序过程。排序中，当关键字间的比较呈逆序时，需要交换两个记录的位置，使用一个辅助空间完成交换，所以其空间复杂度为 O(1)，排序前后两个关键字 5 的相对次序保持不变，因此冒泡法是稳定的排序。

冒泡排序的算法实现如下。

```
void bubbleSort(RecordType A[ ], int n)
{
 int i,j,k,flag=1;
 RecordType temp;
 for(i=1;i<n;i++)
 {
 for(j=0;j<n-i;j++)
  if(A[j].key>A[j+1].key)
  {
    temp=A[j];
    A[j]=A[j+1];
    A[j+1]=temp;
    flag=0;}
 if(flag==1)
    break;
 }
}
```

在该算法中,外层循环控制排序的执行趟数,内层循环用于控制在一趟冒泡排序中相邻记录间的比较和交换。

有 n 个记录的待排序列进行冒泡排序,算法的时间复杂度依赖于待排序序列的初始特性,有以下几种情况。

(1) 如果初始记录序列为“正序”序列,则只进行一趟排序,记录移动次数为 0,关键字间的比较次数为 n−1。

(2) 如果初始记录序列为“逆序”序列,则进行 n−1 趟排序,每一趟中的比较和交换次数将达到最大,即冒泡排序的最大比较次数、最大移动次数分别为

$$C_{max} = \sum_{i=2}^{n} i-1 = \frac{n(n-1)}{2}$$

$$M_{max} = 3 \times C_{max} = \frac{3n(n-1)}{2}$$

(3) 一般情况下,比较次数小于等于 C_{max},移动次数小于等于 M_{max},因此时间复杂度为 $O(n^2)$。

9.3.2 快速排序


视频讲解

快速排序(Quick Sorting)又称分区交换排序,是对冒泡排序算法的改进,是一种基于分治思想的排序方法。

快速排序的基本思想是:从待排记录序列中任取一个记录 R_i 作为基准(存在不同的基准选取办法)将所有记录分成两个序列分组,使排在 R_i 之前的序列分组的记录关键字都小于等于基准记录的关键字值 R_i.key,排在 R_i 之后的序列分组的记录关键字都大于 R_i.key,形成以 R_i 为分界的两个分组,此时基准记录 R_i 的位置就是它的最终排序位置。此趟排序称为第一趟

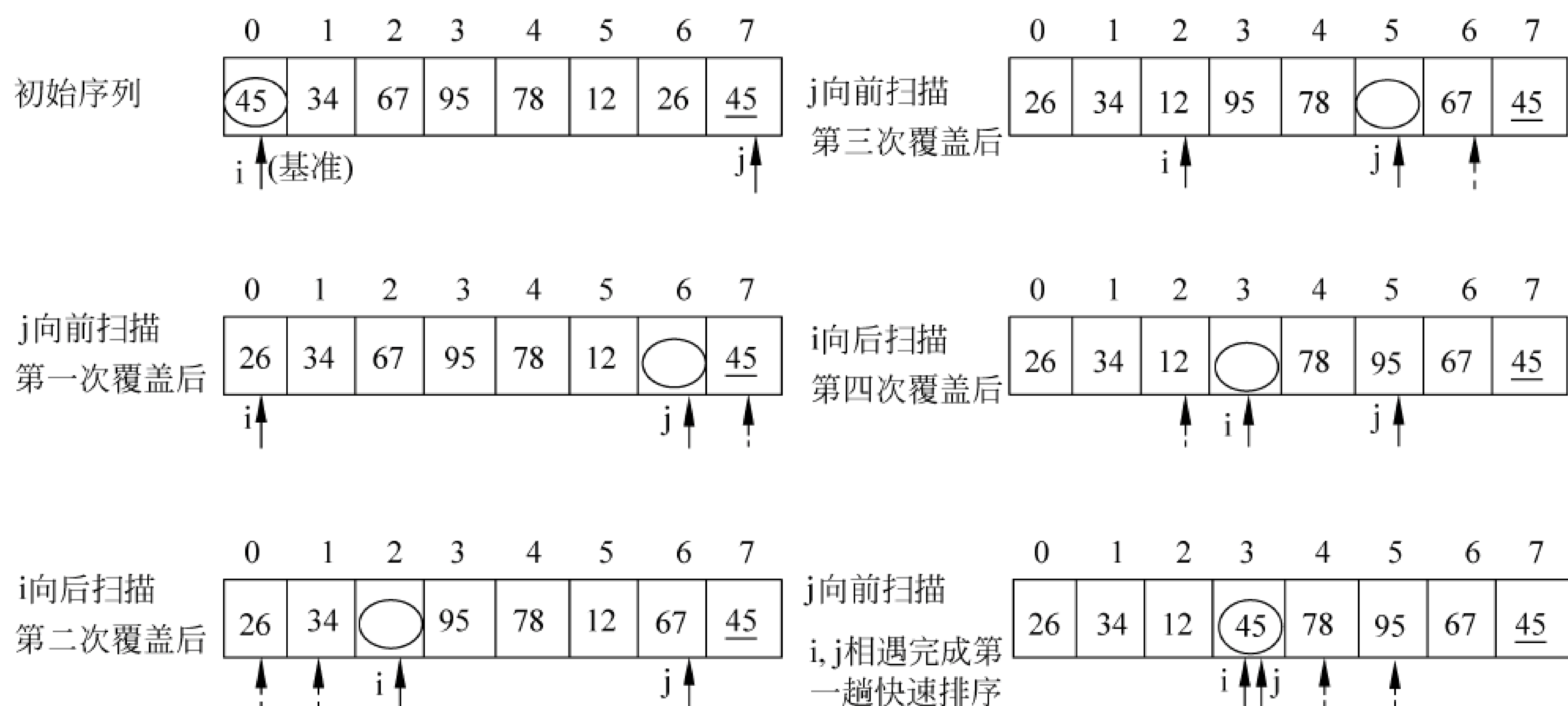
快速排序。然后分别对两个序列分组重复上述过程,直到所有记录排在相应的位置上。

以数组为例,数组 A[p,…,r]被划分成两个子数组 A[p,…,q−1]和 A[q+1,…,r],A[q]为基准,从而使得 A[p,…,q−1]中的每一个元素都小于等于 A[q],而 A[q+1,…,r]中的每个元素都大于等于 A[q]。然后通过递归调用快速排序,对子数组 A[p,…,q−1]和 A[q+1,…,r]进行相同的排序操作。

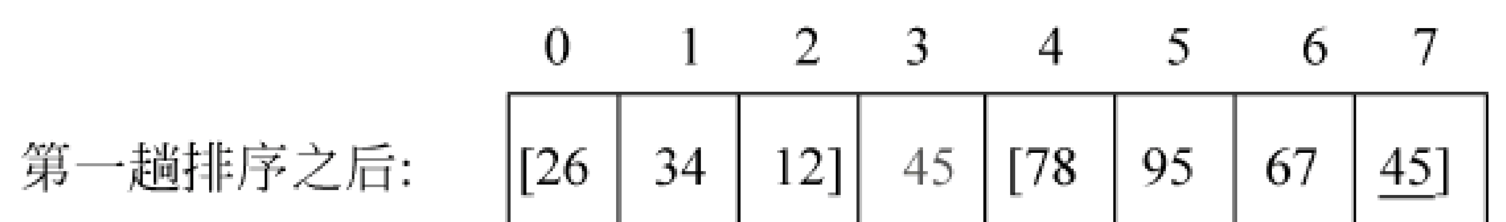
在实际快速排序中,选取基准常用的方法有:

- 选取序列中第一个记录的关键字值作为基准关键字。这种选择方法简单,但是,当序列中的记录已基本有序时,这种选择往往使两个序列分组的长度不均匀,不能改进排序的时间性能。
- 选取序列中间位置记录的关键字值作为基准关键字。
- 比较序列中始端、终端及中间位置上记录的关键字值,并取这 3 个值中居中的一个作为基准关键字。

为了叙述方便,在下面的快速排序中,选取第一个记录的关键字作为基准关键字。假设有 8 个记录的关键字序列{45,34,67,95,78,12,26,<u>45</u>},其快速排序过程如图 9.5 所示。



(a) 快速排序的第一趟



分别进行快速排序: [12] 26 [34]

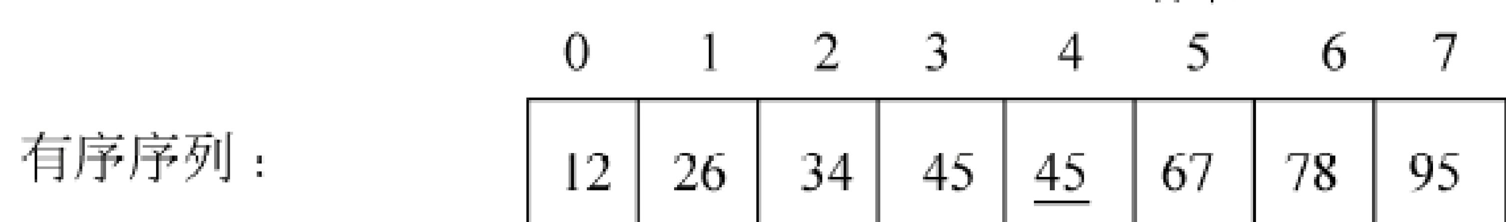
结束 结束

[45 67] 78 [95]

结束

45 [67]

结束



(b) 快速排序的全过程

图 9.5 快速排序过程

快速排序算法实现如下。

```
void QuickSort(RecordType A[ ],int s, int t)
{//对记录序列 A[s…t]进行快速排序
 if(s<t)
 {
  k=Partition(A,s,t);
  QuickSort(A,s,k-1);
  QuickSort(A,k+1,t);
  }
 }
int Partition(RecordType A[ ],int l,int h)
{
//交换记录子序列 A[l..h]中的记录,使基准记录找到最终的位置,并返回其所在位置
 int i=l,j=h;
 KeyType x;
 A[0]=A[i];
 x=A[i].key;
 while(i<j)
 { while(i<j&&A[j].key>=x)  j--;
   A[i]=A[j];
   while(i<j&&A[i].key<=x)  i++;
   A[j]=A[i];
 }
A[i]=A[0];
return i;
}
```

快速排序算法的执行时间取决于基准记录的选择。每一趟快速排序的基准可能不同,因此快速排序是**不稳定**的。一趟快速排序算法的时间复杂度为 O(n)。下面分几种情况讨论整个快速排序算法需要排序的趟数。

- 在理想情况下,每次排序时选取的记录关键字值都是当前待排序列中的"中值"记录,那么,该记录的排序终止位置应在该序列的中间,这样就把原来的子序列分解成了两个长度大致相等的更小的子序列,在这种情况下,排序的速度最快。设完成 n 个记录待排序列所需的比较次数为 C(n),则有

$$C(n) \leqslant n + 2C\left(\frac{n}{2}\right) \leqslant 2n + 4C\left(\frac{n}{4}\right) \leqslant \cdots \leqslant kn + nC(1)$$

其中,k 是序列的分解次数。

若 n 为 2 的幂次值且每次分解都是等长的,则分解过程可用一棵满二叉树描述,分解次数等于树的深度 $k=\log_2 n$,因此有

$$C(n) \leqslant n\log_2 n + nC(1) = O(n\log_2 n)$$

整个算法的时间复杂度为 $O(n\log_2 n)$。

- 在极端情况下,即每次选取的"基准"都是当前分组序列中关键字最小(或最大)的

值，划分的结果是基准的前边（或右边）为空，即把原来的分组序列分解成一个空序列和一个长度为原来序列长度减 1 的子序列。总的比较次数达到最大值：

$$C_{max} = \sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} = O(n^2)$$

如果初始记录序列已为升序或降序排列，并且选取的基准记录又是该序列中的最大或最小值，这时的快速排序就变成了“慢速排序”，整个算法的时间复杂度为 $O(n^2)$。为了避免这种情况发生，可修改上面的排序算法，在每趟排序之前比较当前序列的第一、最后和中间记录的关键字，取关键字居中的一个记录作为基准值调换到第一个记录的位置。

- 一般情况下，序列中各记录关键字的分布是随机的，因而可以认为快速排序算法的平均时间复杂度为 $O(n\log_2 n)$。实验证明，当 n 较大时，快速排序是目前被认为最好的一种内部排序方法。

在算法实现中须设置一个栈的存储空间实现递归，栈的大小取决于递归深度，最多不会超过 n。若每次都选较长的分组序列进栈，而处理较短的分组序列，则递归深度最多不会超过 $\log_2 n$，因此快速排序需要的辅助存储空间为 $O(\log_2 n)$。快速排序是不稳定排序，对于有相同关键字的记录，排序后有可能颠倒位置。

9.4 选择排序

选择排序的基本思想是：不断从待排记录序列中选出关键字最小的记录（或最大的记录）插到已排序记录序列的后面（前面），直到 n 个记录全部插入已排序记录序列中。本节主要介绍简单选择排序和堆排序，同时对锦标赛排序也做了简要介绍。

9.4.1 简单选择排序

简单选择排序（Simple Selection Sort）也称直接选择排序，是选择排序中最简单直观的一种方法。其基本操作思想为

step1：每次从待排记录序列中选出关键字最小的记录。

step2：将最小的记录与待排记录序列第一位置的记录交换后，再将其“插入”已排序记录序列后面（初始为空）。

step3：不断重复过程 **step1** 和 **step2**，即不断地从待排记录序列剩下的记录中选出关键字最小的记录与该区第 1 位置的记录交换（该区第 1 个位置不断后移，该区记录逐渐减少），然后把第 1 位置的记录不断“插入”已排序记录序列之后。经过 n－1 次的选择和多次交换后，$R_1 \sim R_n$ 就排成了有序序列，整个排序过程结束。具有 n 个记录的待排记录序列要做 n－1 次的选择和交换，才能成为有序表。

采用简单选择排序对以下 8 个记录进行排序，图 9.6 是简单选择排序的过程示意图。图中，[ ]中的数据表示待排记录序列的关键字。

简单选择排序的算法实现如下。

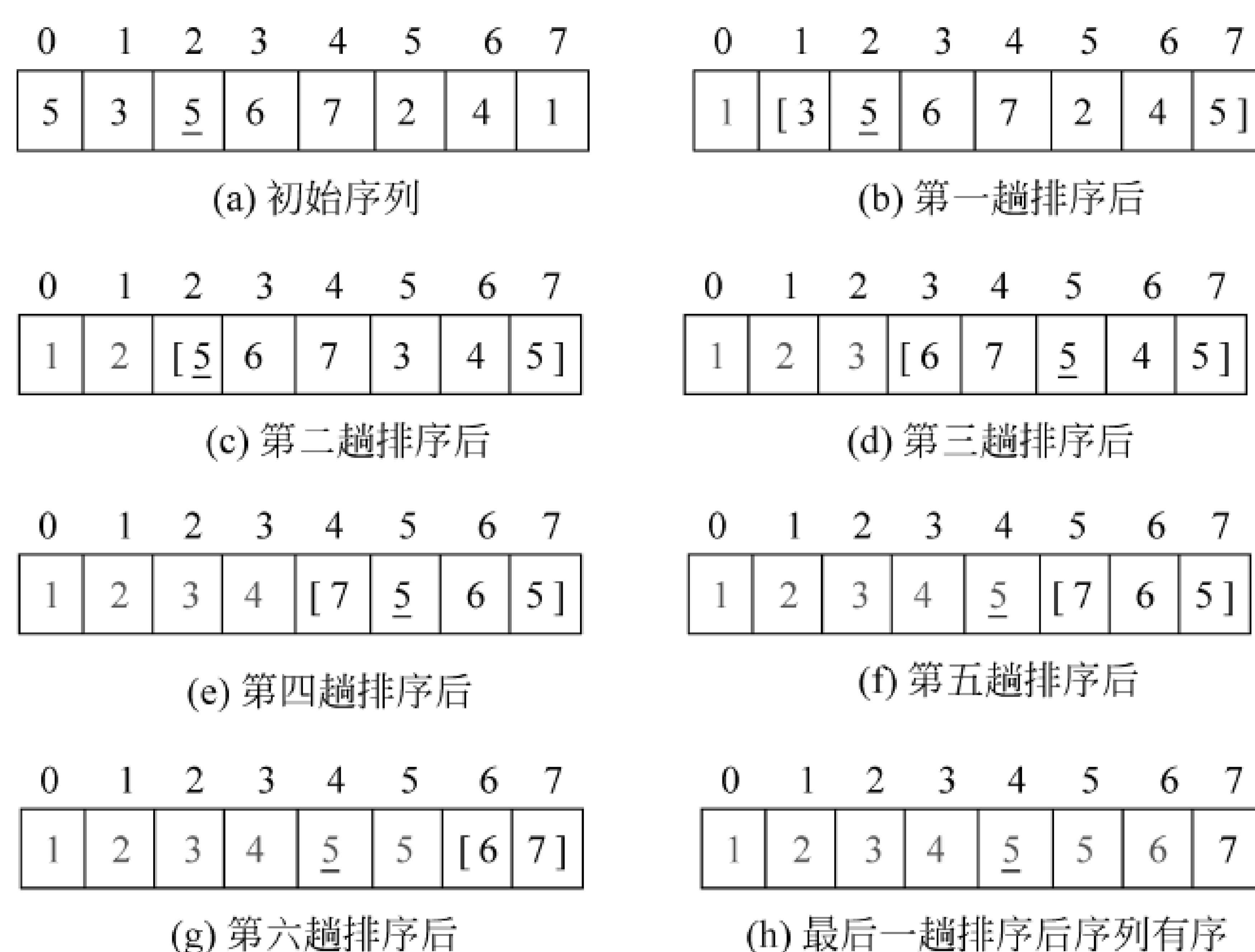


图 9.6 简单选择排序的过程示意图

```
void SelectSort(RecordType A[ ],int n)
{
 int i,j;
 RecordType temp;
 for(i=1;i<n;i++)
 {
   k=i;
   for(j=i+1;j<=n;j++)
     if(A[j].key<A[k].key)
     k=j;
  if(i!=k)
  { temp=A[i];
    A[i]=A[k];
    A[k]=temp;
  }
 }
}
```

简单选择排序算法的关键字比较次数与记录的初始排列无关。假定整个序列表有 n 个记录,总共需要 n−1 趟的选择;第 i (i=1,2,…,n−1)趟选择具有最小关键字记录所需要的比较次数是 n−i−1 次,总的关键字比较次数为

$$(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}$$

记录的移动次数与其初始排列有关。当这组记录的初始状态是按关键字从小到大有序时,每一趟选择后都不需要进行交换,记录的总移动次数为 0,这是最好的情况;而最坏的情况是每一趟选择后都要进行交换,一趟交换需要移动记录 3 次。总的记录移动次数为 3(n−1)。所以,简单选择排序的时间复杂度为 $O(n^2)$。

简单选择排序算法只需要一个临时单元用作交换,因此空间复杂度为 O(1)。由于在直

接选择排序过程中存在不相邻记录之间的互换，可能会改变具有相同关键字记录的相对位置，所以该算法是**不稳定**排序。

9.4.2 锦标赛排序

锦标赛排序也称树形选择排序，是一种按照锦标赛的思想进行选择的排序方法，该方法是在简单选择排序方法上的改进。通过上文的分析我们知道，简单选择排序的时间大部分花费在关键值的比较上面。锦标赛排序即利用树结构保存了前面的比较结果。它的基本思想与体育淘汰赛类似，首先取 n 个元素的关键字进行两两比较，得到$\lfloor n/2 \rfloor$个比较的优胜者，在下一次比较时，直接利用前面的比较结果再两两进行比较。如此重复，直到选出一个关键字最小的对象为止。这类似于比赛中甲、乙、丙 3 队参赛，如果乙胜丙，甲胜乙，则认为甲必能胜丙。这一操作使时间复杂度由 $O(n^2)$降到 $O(n\log_2 n)$。

在此可以通过一个图更好地理解锦标赛排序。假设数组 A＝{3,4,1,6,2,8,7,9}。首先需要建立一棵完全二叉树。注意，如果数组的长度不是 2 的完整次幂，则需要补一些元素。图 9.7 表示了这一过程，其实数组 A 的元素完全分布在叶子结点上，其他分支结点是为了存储锦标赛的结果。

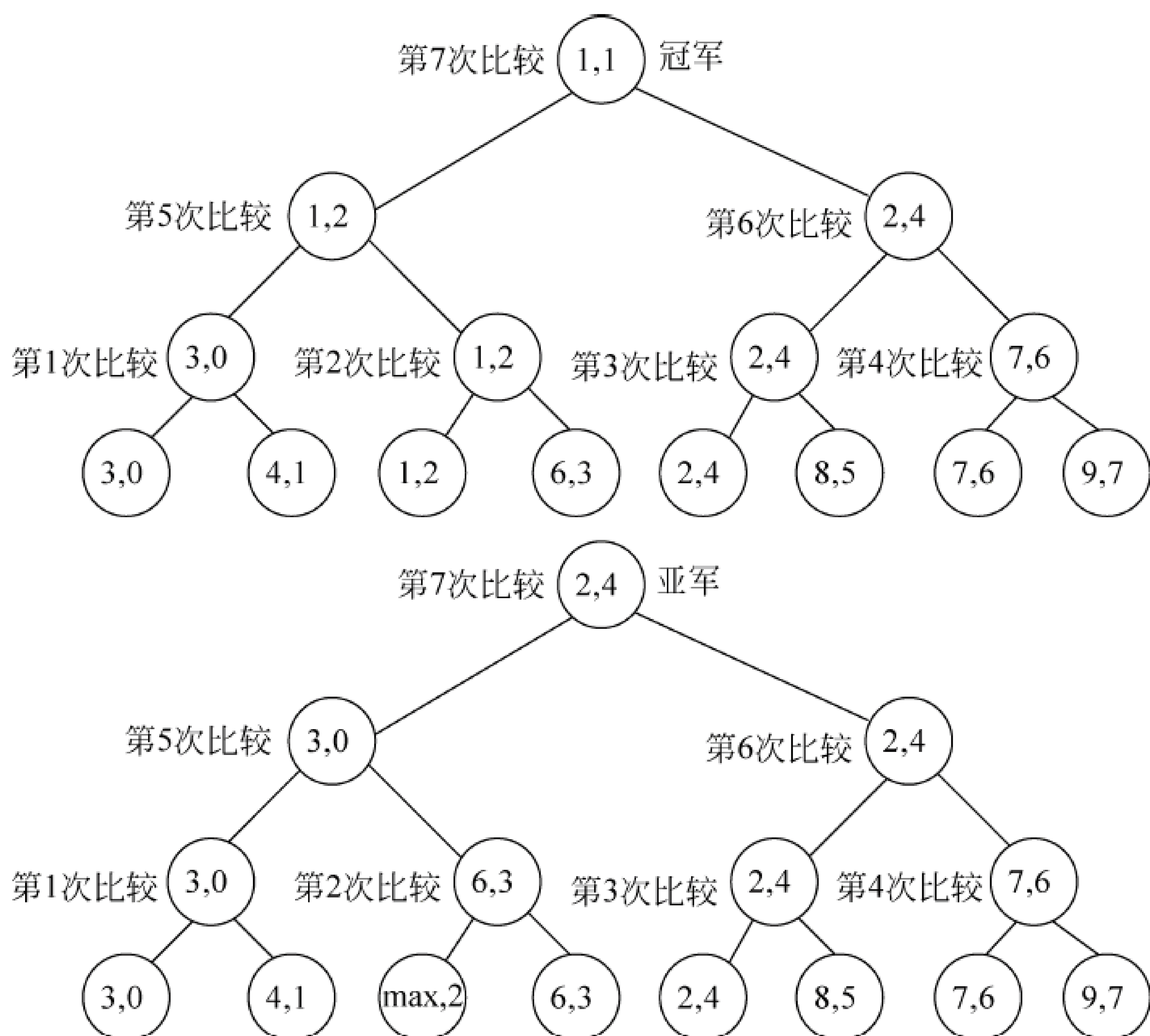


图 9.7 锦标赛排序过程

如图 9.7 所示，对于 n 个记录的锦标赛排序，每选择一个记录仅需要进行$\lfloor \log_2 n \rfloor$次比较。具体做法为：输出“冠军”后，将冠军的叶子结点关键字改为最大 max，继续进行锦标赛排序，直到选出关键字的次小记录为止，如此循环，直到输出全部有序序列。因此，它的时间复杂度为 $O(n\log_2 n)$。这种方法的缺点是对最大值 max 进行了多次比较。

9.4.3 堆排序



视频讲解

如何提高空间效率呢?堆排序(Heap Sort)为我们提供了解决思路。堆排序方法是由 J. Williams 和 Floyd 提出的一种改进方法,它在选择当前最小关键字记录的同时,还保存了本次排序过程产生的比较信息。借助完全二叉树结构进行排序,既具有空间原址性,又拥有 O(nlogn)的时间复杂度,是一种巧妙的排序算法。

堆可以分为两种形式:**最大堆**和**最小堆**。我们给出最大堆的定义,最小堆与其刚好相反。

n 个元素序列$\{k_1, k_2, \cdots, k_n\}$,当且仅当满足如下性质时,称为**最大堆**。

- 这些元素是一棵完全二叉树中的结点,且对于 i=1,2,…,n,k_i 是该完全二叉树中编号为 i 的结点。

$k_i \geqslant k_{2i}, (1 \leqslant i \leqslant \lfloor n/2 \rfloor)$
$k_i \geqslant k_{2i+1}, (1 \leqslant i \leqslant \lfloor n/2 \rfloor)$

从堆的定义可以看出,堆是一棵完全二叉树,其中每一个非终端结点的元素均大于等于(或小于等于)其左、右孩子结点的元素值。图 9.8 给出的示例对应的元素序列分别为{92,84,25,36,14,07}和{15,39,23,87,44,31,52,90},{17,25,23,33,62,96,87,15}。

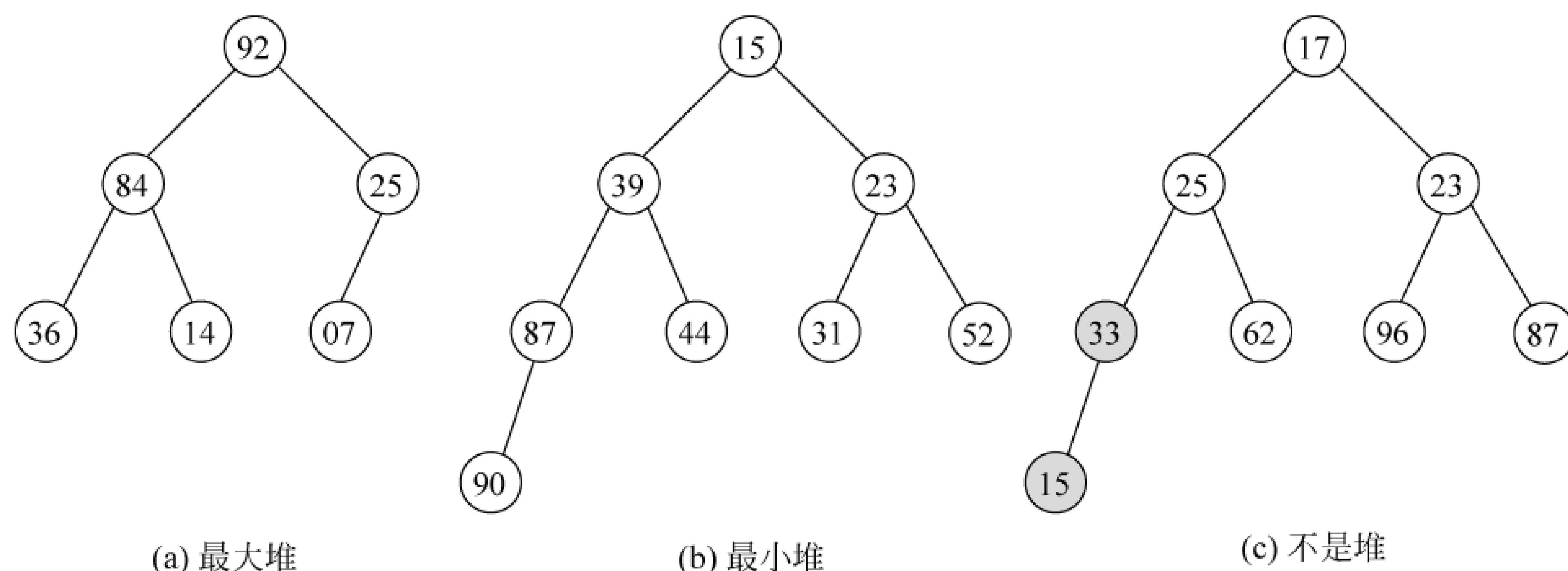


图 9.8 堆定义判定

根据堆的定义,可以推出堆的两个性质。

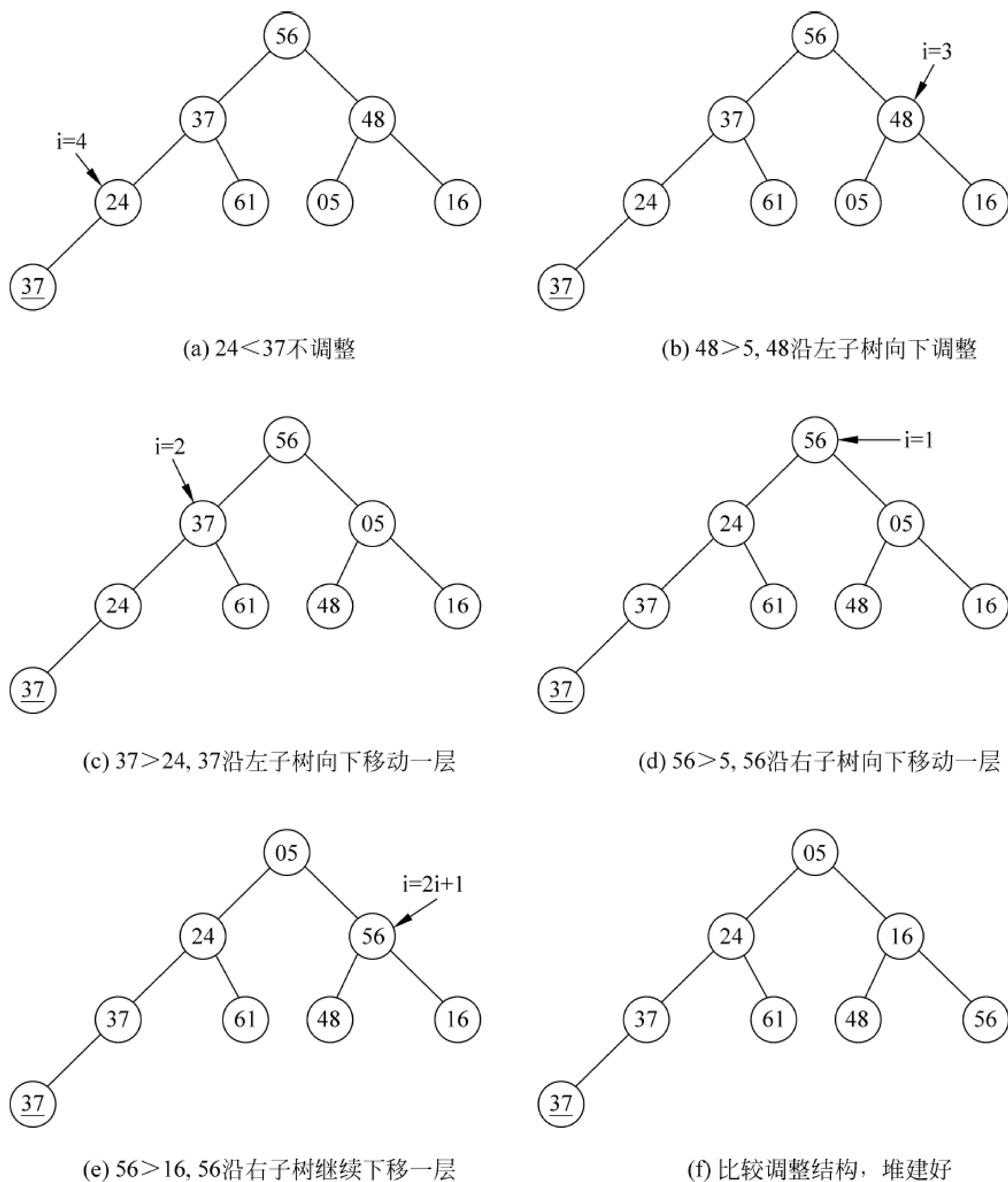
- 堆的根结点是堆中元素值最小(或最大)的结点,称为堆顶元素。
- 从根结点到每个叶子结点的路径上,元素的排序序列都是递增(或递减)有序的。

堆排序的基本思想是:对一组待排序记录,首先把它们的关键字按堆定义排列成一个序列(称为**初始建堆**),堆顶元素为最小关键字的记录,将堆顶元素输出;然后对剩余的记录再建堆,得到次最小关键字记录;如此反复进行,直到全部记录有序为止,这个过程称为堆排序。

那么,如何将一个无序序列建成一个堆?具体做法是:把待排序记录存放在数组 A[1,…,n]中,将 R 看作一棵完全二叉树,每个结点表示一个记录,将第一个记录 A[1]作为二叉树的根,以下各记录 A[2,…,n]依次逐层从左到右顺序排列,构成一棵完全二叉树,任意结点 A[i]的左孩子是 A[2i],右孩子是 A[2i+1],双亲是 A[i/2]。将待排序的所有记录放到一

棵完全二叉树的各个结点中(注意：这时的完全二叉树不一定具备堆的特征)。此时所有 $i>\lfloor n/2 \rfloor$的结点 A[i]都没有孩子结点(即为叶子结点),因此以 A[i]为根的子树已经是堆。从 $i=\lfloor n/2 \rfloor$的结点 A[i]开始,比较根结点与左、右孩子的关键字值,若根结点的值大于左、右孩子中的较小者,则交换根结点和值较小孩子的位置,即把根结点下移,然后根结点继续和新的孩子结点比较,如此一层一层地递归下去,直到根结点下移到某一位置时,它的左、右子结点的值都大于它的值或者已成为叶子结点,这个过程称为“筛选”。从一个无序序列建堆的过程就是一个反复“筛选”的过程,“筛选”需要从 $i=\lfloor n/2 \rfloor$的结点 A[i]开始,直至结点 A[1]结束。

例如,有一个 8 个元素的无序序列{56,37,48,24,61,05,16,37},它对应的完全二叉树及其建堆过程如图 9.9 所示。因为 n=8,n/2=4,所以从第 4 个结点起至第一个结点止,依次对每一个结点进行“筛选”。



(a) 24<37不调整

(b) 48>5, 48沿左子树向下调整

(c) 37>24, 37沿左子树向下移动一层

(d) 56>5, 56沿右子树向下移动一层

(e) 56>16, 56沿右子树继续下移一层

(f) 比较调整结构，堆建好

图 9.9　初始化建堆

如图 9.9 所示,初始化建堆过程就是从最后一个分支结点开始重复筛选的过程。

筛选算法实现如下。

```
void Sift(RecordType A[ ],int i,int n)
{
 int j=2*i;
A[0]=A[i];                          //将 A[i]保存在临时单元中
while(j<=n)
{ if((j<n)&&(A[j].key>A[j+1].key))
   ++j;                             //选择左、右孩子中的最小者
  if(A[0].key>A[j].key)             //当前结点大于左、右孩子的最小者
  { A[i]=A[j];
    i=j; j=2*i; }
  else                              //当前结点不大于左、右孩子
    break;
 }
 A[i]=A[0];                         //被筛选结点放到最终合适的位置上
}
```

建初始堆的过程描述为

```
for(i=n/2;i>0;--i)
Sift(A,i,n);
```

在输出堆顶记录之后,如何调整剩余记录成为一个新的堆?由堆的定义可知,在输出堆顶记录之后,以根结点的左、右孩子为根的子树仍然为堆。为了把剩余的记录建成一个新堆,可以将堆的最后一个记录放到堆顶位置作为根结点,形成一个新的完全二叉树。该完全二叉树不是一个堆,但根结点的左、右子树均为堆。此时只需将根结点由上至下"筛选"到合适的位置,使它的左、右孩子的关键字值都大于它的值,至此就完成了新堆的建立。

调整堆,使其保持堆的特性的过程为

```
for(j=n;j>1;--j)
{
A[0]=A[1];
A[1]=A[j];
A[j]=A[0];
Sift(A,1,j-1);
 }
```

对于已建好的堆,可以采用下面两个步骤进行排序。

step1:输出堆顶元素,将堆顶元素(第一个记录)与当前堆的最后一个记录对调。

step2:调整堆,将输出根结点之后的新完全二叉树调整为堆。

step3:不断地输出堆顶元素,又不断地把剩余的元素建成新堆,直到所有的记录都变成堆顶元素输出。

堆排序的算法描述如下。

```
void HeapSort(RecordType A[ ],int n)
{int j;
 for(j=n/2;j>0;--j)                    //建初始堆
   Sift(A,j,n);
 for(j=n;j>1;--j)
 {                                     //进行 n-1 趟排序
     A[0]=A[1];                        //将堆顶元素与堆中的最后一个元素交换
     A[1]=A[j]; A[j]=A[0];
     Sift(A,1,j-1);                    //将 A[1]..A[j-1]调整为堆
   }
 }
```

下面对无序序列{56,37,48,24,61,05,16,37}进行堆排序，堆排序过程如图 9.10 所示。

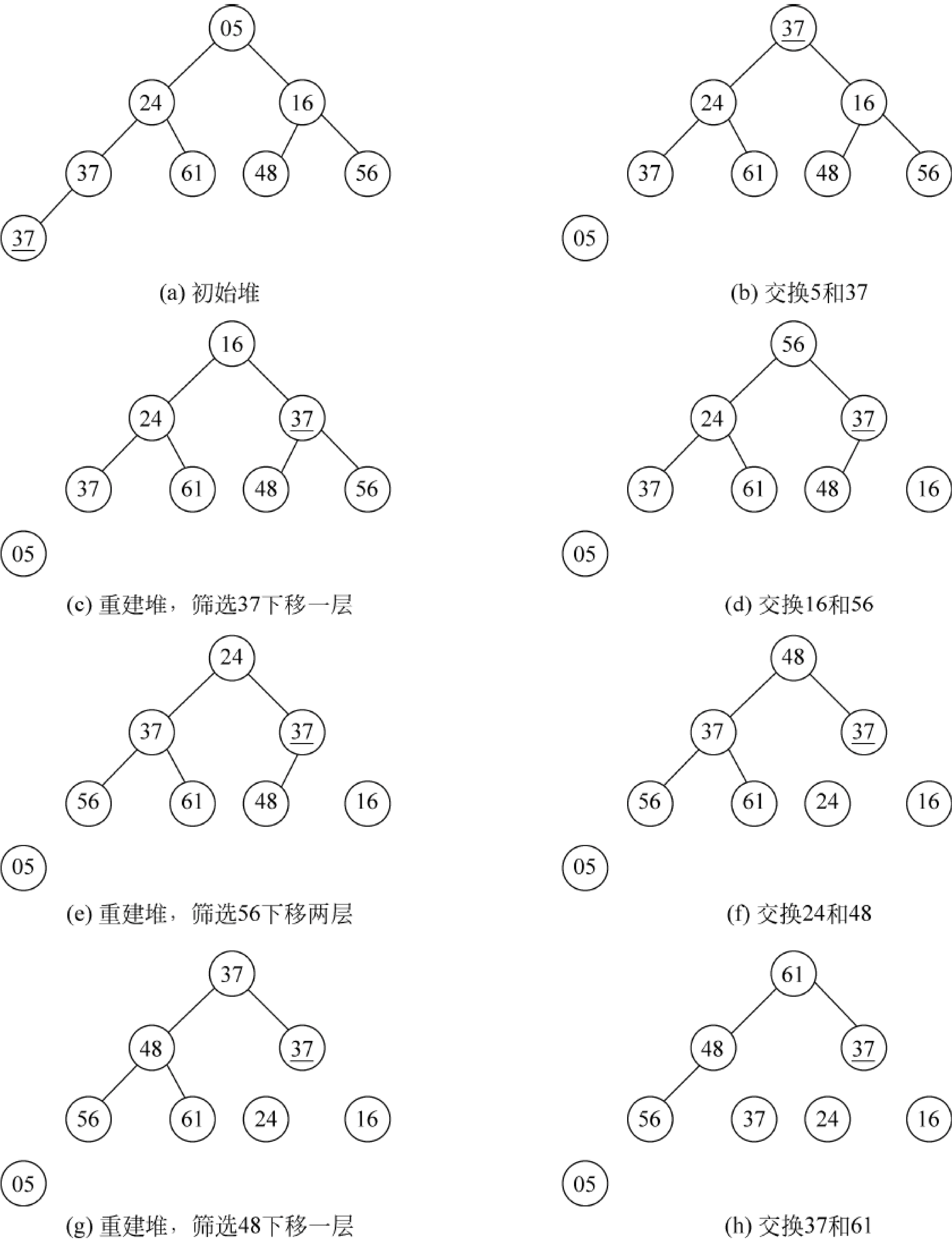


图 9.10　堆排序过程

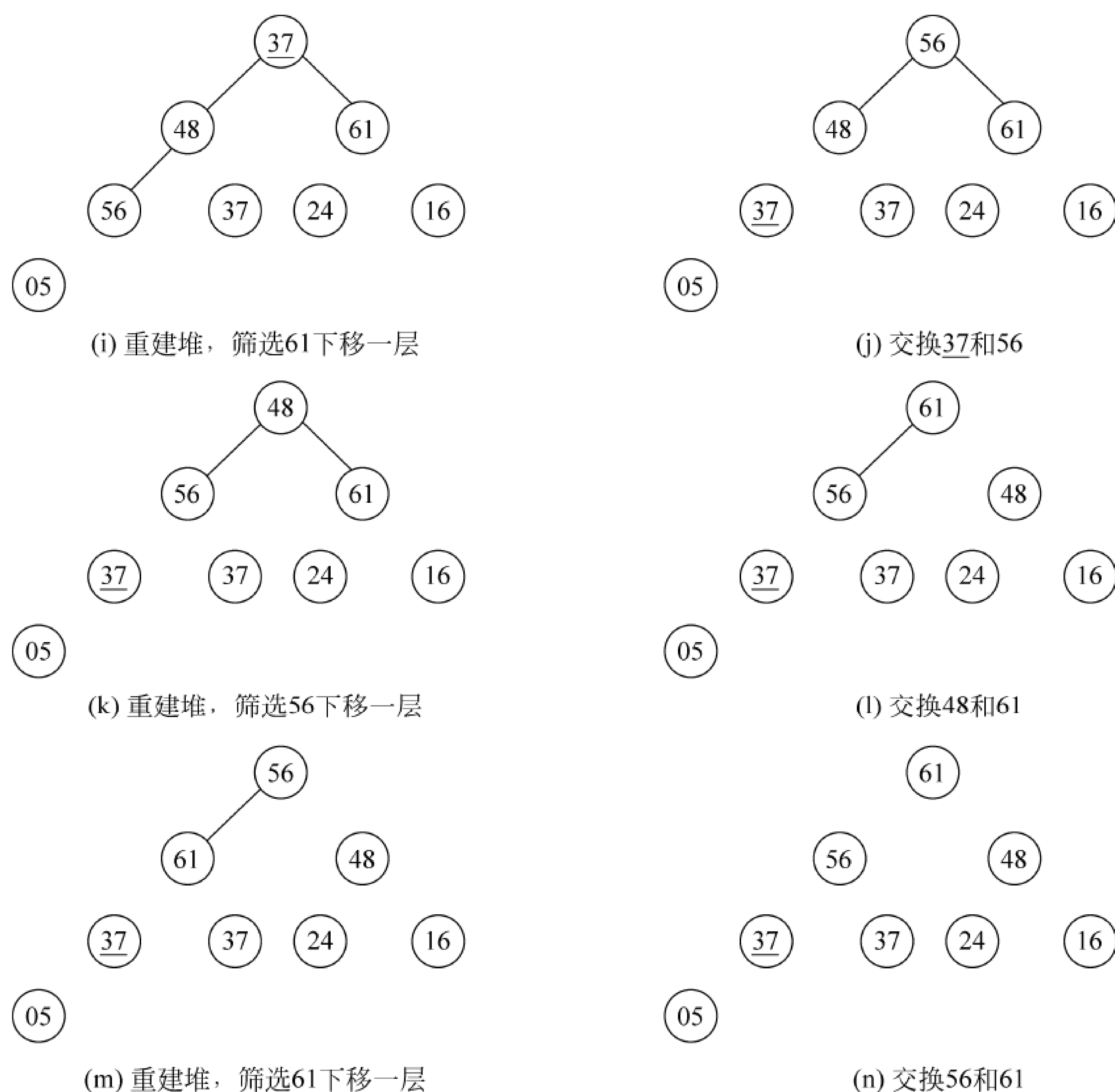


图 9.10 （续）

由图 9.10 可以再一次看到，堆排序算法主要由建立初始堆和反复重建堆两部分构成，它们均通过调用 Sift()函数实现。假设具有 n 个记录的初始序列对应的完全二叉树的深度为 $h=\lfloor \log_2 n \rfloor+1$，则在建立初始堆时，对每一个非叶子结点都要从上到下做“筛选”，建立初始堆的总比较次数 C_1 为：$C_1 \leqslant 4n$，其时间复杂度为 O(n)。n 个结点完全二叉树的深度为 $h=\lfloor \log_2 n \rfloor+1$，n−1 次建新堆的总比较次数为 C_2：$C_2(n) \leqslant 2(\lfloor \log_2 n \rfloor+\lfloor \log_2(n-1) \rfloor+\cdots+\log_2 2) \leqslant 2n \times \log_2 2$。所以，堆排序整体所需的关键字比较的总次数是 $C_1+C_2=O(n\log_2 n)$。类似地，可求出堆排序所需的记录移动的总次数为 $O(n\log_2 n)$，因此堆排序的最坏时间复杂度为 $O(n\log_2 n)$。堆排序算法一般适合于待排序记录数比较多的情况。堆排序只需要一个辅助空间，所以空间复杂度为 O(1)。堆排序也是**不稳定**排序。

9.5　二路归并排序

归并排序也是一种常用的排序方法。“归并”的含义是将两个或两个以上的有序表合并成一个新的有序表。图 9.11 为两组有序表的归并，有序表{4,25,34,56,69,74}和{15,26,34,47,52}通过归并把它们合并成一个有序表{4,15,25,26,34,34,47,52,56,69,74}。

二路归并排序的基本思想是：将有 n 个记录的待排序列看作 n 个有序子表，每个有序子表的长度为 1，然后从第一个有序子表开始，把相邻的两个有序子表两两合并，得到 n/2 个长度为 2 或 1 的有序子表（当有序子表的个数为奇数时，最后一组合并得到的有序子表长度为 1），这一过程称为一趟归并排序。再将有序子表两两归并，如此反复，直到得到一个长度为 n 的有序表为止。上述每趟归并排序都需要将相邻的两个有序子表两两合并成一个有序表，这种归并方法称为二路归并排序。

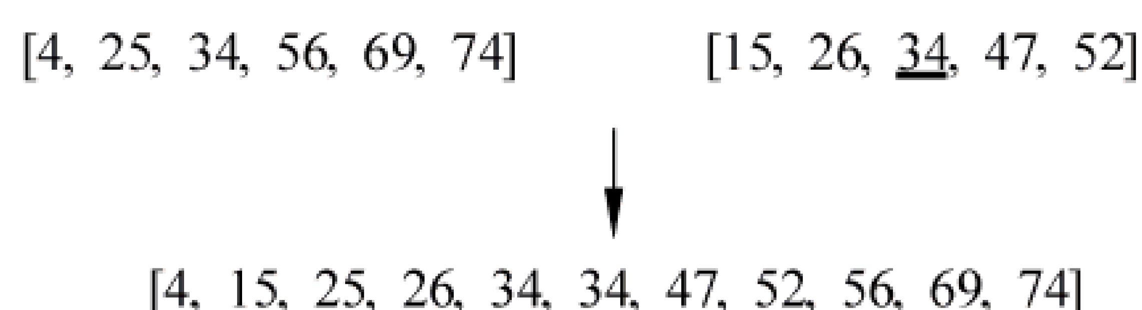


图 9.11　二路归并排序

要想实现一趟归并排序，首先要实现两个有序表的合并算法 Merge()。设线性表 R[low，…，m]和 R[m+1，…，high]是两个已排序的有序表，存放在同一数组中相邻的位置上，将它们合并到一个数组 R1 中，合并过程如下。

step1：比较有序表 R[low，…，m]与 R[m+1，…，high]的第一个记录，将其中关键字值较小的记录移入表 R1（如果关键字值相同，可将 R[low，…，m]的第一个记录移入 R1 中）。

step2：将关键字值较小的记录所在线性表的长度减 1，并将其后继记录作为该线性表的第一个记录。

step3：反复执行上述过程，直到线性表 R[low，…，m]或 R[m+1，…，high]之一成为空表，然后将非空表中剩余的记录移入 R1 中，此时 R1 成为一个有序表。

两个有序子表归并的算法实现如下。

```
void Merge(RecordType R[ ],RecordType R1[ ],int low,int m,int high)
{ //R[low…m]和 R[m+1…high]是两个有序表
 int i=low, j=m+1, k=low;
 //k 是 R1 的下标,i,j 分别为 R[low…m]和 R[m+1…high]的下标
 while(i<=m&&j<=high)
{
 //在 R[low…m]和 R[m+1…high]均未扫描完时循环
 if(R[i].key<=R[j].key)
  { //将 R[low…m]中的记录放入 R1 中
     R1[k]=R[i]; i++; k++;
   }
 else
 {                         //将 R[m+1…high]中的记录放入 R1 中
     R1[k]=R[j]; j++; k++;
  }
 }
while(i<=m)
{                          //将 R[low…m]的余下部分复制到 R1
 R1[k]=R[i];
 i++; k++;
  }
while(j<=high)
{                          //将 R[m+1…high]的余下部分复制到 R1
 R1[k]=R[j];
 j++;k++;
```

```
    }
  }
```

一趟归并排序的算法 MergePass()调用⌊n/(2 * length)⌋次归并算法 Merge(),将 R[1,…,n]中前后相邻且长度为 length 的有序子表进行两两归并,得到前后相邻且长度为 2 * length 的有序表,并存放在 R1[1,…,n]中。如果 n 不是 2 * length 的整数倍,则可出现两种情况:一种情况是,剩下一个长度为 length 的有序子表和一个长度小于 length 的子表,合并之后其有序表的长度小于 2 * length;另一种情况是,只剩下一个子表,其长度小于等于 length,此时不调用算法 Merge(),只将其直接放入数组 R1 中,准备进行下一趟归并排序。

一趟归并排序的算法描述如下。

```
void MergePass(RecordType R[ ],RecordType R1[ ],int length, int n)
{int i=0,j;
 while(i+2 * length-1<n){
 Merge(R,R1,i,i+length-1,i+2 * length-1);
 i=i+2 * length;                        //归并长度为 length 的两相邻有序子表
  }
  if(i+length-1<n-1)                    //余下两个有序子表,其中一个长度小于 length
    Merge(R,R1,i,i+length-1,n-1);       //归并两个有序表
  else
    for(j=i;j<n;j++)                    //剩下一个有序子表,其长度小于 length
 R1[j]=R[j];
}
```

二路归并排序算法 MergeSort()需要由多趟归并过程实现。第一趟 length=1,以后每执行一趟归并后将 length 加倍。第一趟归并的结果存放在 R1 中;第二趟将数组 R1 中的有序子表两两合并,结果存放在数组 R 中;如此反复进行。为使最终排序结果存放在数组 R 中,进行归并的趟数必须是偶数。因此,当只需奇数趟归并即可完成排序时,应再进行一趟归并,此时只剩下一个长度不大于 length 的有序表,直接从数组 R1 复制到 R 中即可。

假设初始序列为{23,56,42,37,15,84,72,27,18},用二路归并排序法排序后结果为{15,18,23,27,37,42,56,72,84},整个归并过程如图 9.12 所示。

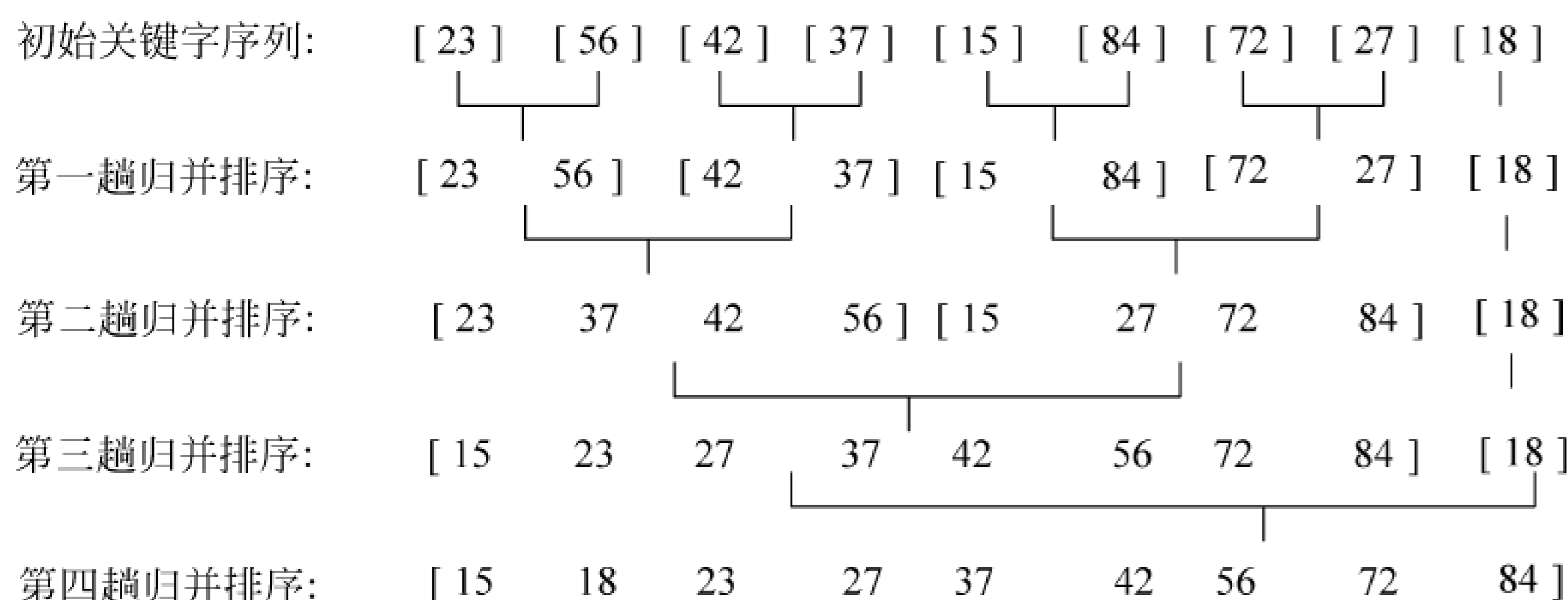


图 9.12 整个归并过程

归并算法实现如下。

```
void MergeSort(RecordType R[ ], int n)
{int length=1;
 while (length<n){
  MergePass(R, R1, length, n);
  length=2 * length;
  MergePass(R1, R, length, n);
  length=2 * length;
 }
}
```

显然，n 个记录进行二路归并排序时，归并的趟数为 $O(n\log_2 n)$，每趟归并中，关键字的比较次数不超过 n，因此，二路归并排序的时间复杂度为 $O(n\log_2 n)$。对序列进行归并排序时，除采用二路归并排序外，还可以采用多路归并排序方法。归并排序需要的辅助空间 R1 与待排序记录的数量相等，因此，二路归并排序的空间复杂度为 $O(n)$，这是常用的排序方法中空间复杂度最差的一种排序方法。另外，从排序的稳定性看，二路归并排序是一种**稳定**的排序方法。

9.6 基数排序

基数排序是和前面所述各类排序方法完全不同的一种排序方法。**基数排序**(Radix Sort)是一种借助多关键字排序的思想对单逻辑关键字进行排序的方法，即先将关键字分解成若干部分，然后通过对各部分关键字的分别排序，最终完成对全部记录的排序。

基数排序首先把每个关键字看作为一个 d 元组：$K_i=(K_i^0, K_i^1, \cdots, K_i^{d-1})$

其中，$C_0 \leqslant K_i^j \leqslant C_{r-1}$ $(1\leqslant i\leqslant n, 0\leqslant j\leqslant d-1)$，r 称为基数。设置 r 个桶，排序时先按 K_i^{d-1} 从大到小将记录**分配**到 r 个桶中，然后依次**收集**这些记录，称之为一趟基数排序。再按 K_i^{d-2} 从大到小将记录分配到 r 个桶中，如此反复，直到对 K_i^0 分配和收集，得到的便是排好序的序列。

基数 r 的选择和关键字的分解法因关键字的类型而异。关键字为十进制整数时，$r=10, C_0=0, C_{r-1}=9$。关键字的每一位取值为 $0\leqslant K_i^j\leqslant 9$，d 为关键字的最大位数。关键字为二进制数时，$r=2, C_0=0, C_{r-1}=1$，关键字的每一位取值为 0 或 1，d 为关键字的最大位数。关键字为字母串时，$r=26$，$C_0=$'A'，$C_{r-1}=$'Z'，关键字的每一位取值为'A'$\leqslant K_i^j \leqslant$'Z'，d 为关键字中字母的最大长度。

基数排序时，为了实现记录的分配和收集，可以设置 r 个队列，排序前均为空队列，分配时将记录分别插(分配)到各自的队列中，收集时将这些队列中的记录排列在一起。使用数组 F[ ]和 E[ ]分别保存各个队列的头、尾指针。设置数组 R 存放待排序记录序列，并令表头结点 head 指向第一个记录，R 数组元素的类型描述为

```
typedef struct dataType
{char key[d];                    //记录中的关键字
```

```
   struct dataType  * next;                    //指向下一个记录的下标
    elemtype otherelement;                     //记录中的其他数据
}srecord;
```

基数排序算法描述为

```
void RadixSort(srecord  * head,srecord * F[ ],srecord * E[ ],int d,int r)
{//head 是指向待排序记录链表的头指针,r 为基数,d 为每个关键字的最大位数
   int i,j,k;
 srecord  * p, * q;
 for(i=0;i<d;i++)                          //循环 d 次,对各位进行分配和收集
 { for(j=0;j<r;j++)                        //清空保存各个队列头、尾指针的数组
    { F[j]=NULL;
      E[j]=NULL;
     }
 p=head;
 while(p!=NULL)                            //进行待排序记录的分配
{ k=(p->key[d-1-i])-'0'};
   //取出关键字的(d-1-i)位的值,用于判断将当前记录链到哪个队列
   if(F[k]==NULL)                          //将记录添到第 k 个队列尾部
     F[k]=p;
   else
     E[k]->next=p;
     E[k]=p;                               //修改尾指针
     p=p->next;
       }
    head=NULL;                             //head 作为收集新记录链表的头指针
    q=NULL;                                //q 作为新记录链表的尾指针
   for(j=0;j<r;j++)                        //收集按关键字(d-1-i)位分配的记录
   { if(F[j]!=NULL)
     { if(head!=NULL)
         q->next=F[j];                     //将第 j 个"桶"链接到 head 链表中
       else
         head=F[j];
       q=E[j];
         }
       }
 q->next=NULL;
 }
}
```

上述算法中,由 while 循环完成记录的分配,每个记录应存放到哪个队列中与关键字(d-1-i)位的取值有关,关键字(d-1-i)位的值通过语句 p->key[(d-1-i)]-‘0’获得。由内层第二个 for 循环完成对已分配记录的收集。

设待排序序列中有 10 个记录,其关键字分别 231,144,037,572,006,249,528,134,065,152,使用基数排序法进行排序,过程如图 9.13 所示。

关键字是十进制整数,r=10,d=3。第一趟分配是对关键字的个位数进行的,将链表中

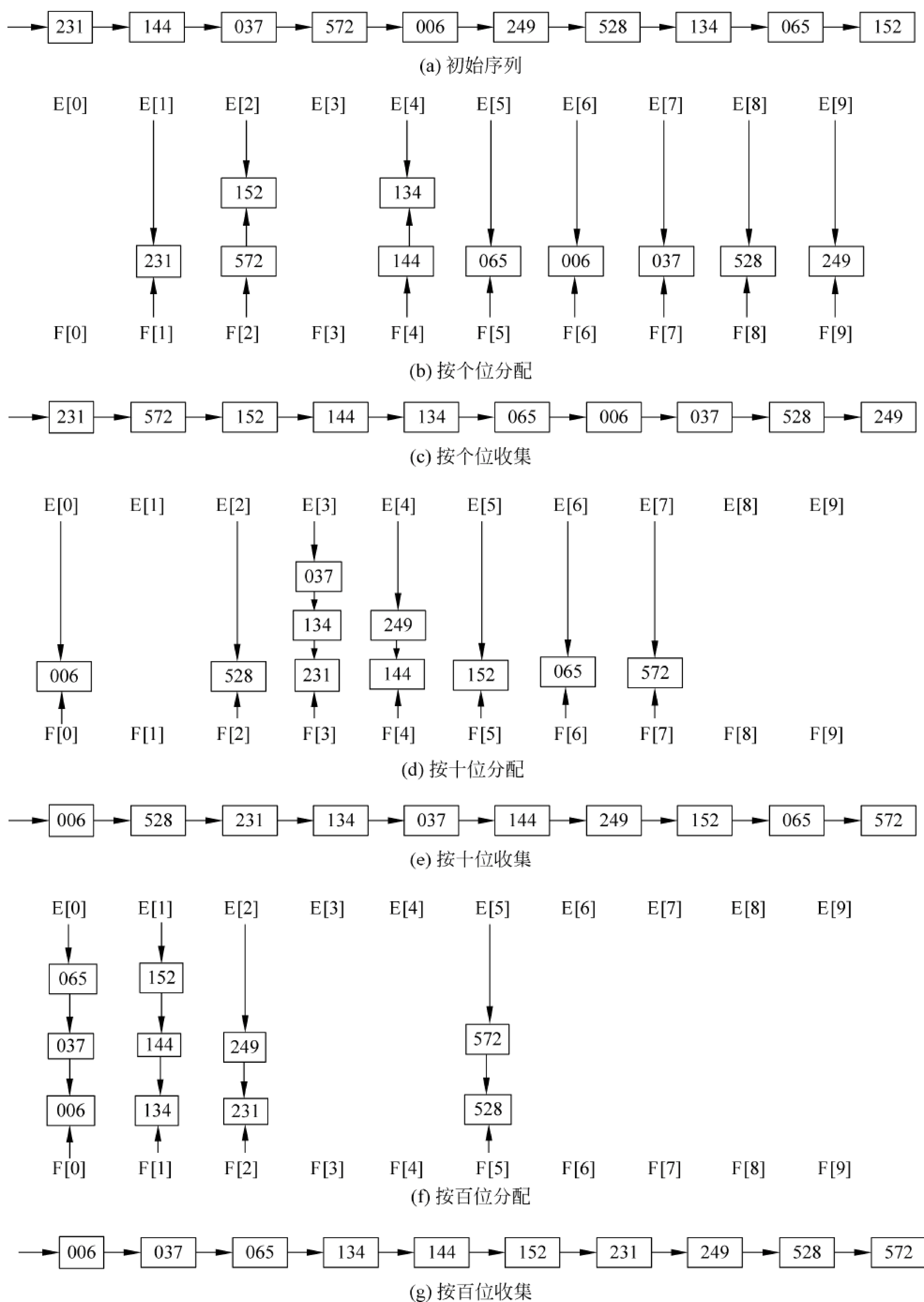


图 9.13　基数排序过程

的记录分配至 10 个队列,每个队列中的记录关键字的个位数相同,如图 9.13(b)所示,其中 F[i]和 E[i]分别为第 i 个队列的头指针和尾指针;第一趟收集是改变所有非空队列的队尾记录的指针域,令其指向下一个非空队列的头指针,重新将 10 个队列中的记录链接成一个

链表,如图 9.13(c)所示;第二趟分配,第二趟收集及第三趟分配和第三趟收集分别是对关键字的十位数和百位数进行的,其过程和个位数相同,如图 9.13(d)~(g)所示,至此排序完毕。

基数排序的执行时间取决于记录关键字 K_i 的最大位数 d。基数排序算法对待排序列中的记录共进行 d 趟分配和收集过程。每趟排序,分配时间为 O(n),收集时间为 O(r),因此一趟基数排序的时间为 O(n+r)。经过 d 趟排序的总时间为 O(d×(n+r))。一般情况下,当 n 很大,d 较小时,此算法很有效。基数排序需要额外设置存放 r 个队列指针的数组,因此空间复杂度为 O(n+r)。从排序的稳定性看,基数排序是一种**稳定**的排序方法。

9.7 内部排序方法的比较

下面是对各种排序方法的比较。

- 插入排序的原理:向有序序列中依次插入无序序列中待排序的记录,直到无序序列为空,对应的有序序列即为排序的结果,其主旨是“插入”。
- 交换排序的原理:先比较大小,如果逆序,就进行交换,直到有序。其主旨是“若逆序,就交换”。
- 选择排序的原理:先找关键字最小的记录,再放到已排好序的序列后面,依次选择,直到全部有序,其主旨是“选择”。
- 归并排序的原理:依次对两个有序子序列进行“合并”,直到合并为一个有序序列为止,其主旨是“合并”。
- 基数排序的原理:按待排序记录的关键字的组成成分进行排序,即依次比较各个记录关键字相应“位”的值,并进行排序,直到比较完所有的“位”,即得到一个有序的序列。

各种排序方法的工作原理不同,对应的性能也有很大的差别。下面通过表 9.1 可以看到各排序方法具体的时间性能、空间性能等方面的区别。

表 9.1 内排序方法对比

排序方法	平均时间复杂度	最坏情况时间复杂度	最佳情况时间复杂度	空间复杂度	稳定性	复杂性
直接插入排序	$O(n^2)$	$O(n^2)$	O(n)	O(1)	稳定	简单
希尔排序	$O(n^{1.3})$			O(1)	不稳定	较复杂
冒泡排序	$O(n^2)$	$O(n^2)$	O(n)	O(1)	稳定	简单
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(n\log_2 n)$	不稳定	较复杂
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定	简单
锦标赛排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	O(n)	不稳定	一般
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	O(1)	不稳定	较复杂
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$		稳定	较复杂
基数排序	O[d(n+r)]	O[d(n+r)]	O[d(n+r)]	O(n+r)	稳定	较复杂

9.8 STL 中的排序

在实际应用中，我们多使用标准模板库中的排序函数。C/C++的 STL 中的排序函数见表 9.2。

表 9.2 STL 中的排序函数

函　数　名	功 能 描 述
sort	对给定区间的所有元素进行排序
stable_sort	对给定区间的所有元素进行稳定排序
partial_sort	对给定区间的所有元素部分排序
partial_sort_copy	对给定区间复制并排序
nth_element	找出给定区间的某个位置对应的元素
is_sorted	判断一个区间是否已经排好序
partition	使得符合某个条件的元素放在前面
stable_partition	相对稳定的使得符合某个条件的元素放在前面

其中最常用的便是 sort 函数。其函数声明为

```
#include <algorithm>
template <class RandomIt>
void sort(RandomIt first, RandomIt last);
template <class RandomIt, class Compare>
void sort(RandomIt first, RandomIt last, Compare comp);
```

sort 函数的使用方法非常简单。STL 提供了两种调用方式：一种是使用默认的<操作符比较；一种可以自定义比较函数。它的显著优点为速度快。对 15 万个随机产生的 int 类型无序数排序，冒泡排序、简单选择排序、STL 的 sort 算法速度对比如图 9.14 所示。

```
1  The time is 39.488 (s)
2  The time is 7.91 (s)
3  The time is 0.01 (s)
4
5
6  Process returned 0 (0x0)   execution time : 48.001 s
7  Press any key to continue.
```

图 9.14 算法速度对比结果

由图 9.14 可知，STL 中的 sort 并非只是普通的快速排序，除了对普通的快速排序进行优化，它还结合了插入排序和堆排序。根据不同的数量级别以及不同的情况，能自动选用合适的排序方法。当数据量较大时，采用快速排序，分段递归。一旦分段后的数据量小于某个阈值，为避免递归调用带来过大的额外负荷，便会改用插入排序。如果递归层次过深，有出现最坏情况的倾向，还会改用堆排序。这使得 sort 成为最通用的排序函数。

然而，sort 函数仍然存在一个缺点：不稳定性。实际应用场景中，如给某班学生排序，先把学号作为关键字，然后把成绩作为关键字，我们希望成绩相同的同学仍然按照学号的次序排列，也就是要求排序的稳定性，这时可以使用 stable_sort 函数。stable_sort 内部首先

判断是否有足够的额外空间(如 vector 中的 cap-size()部分),如果有,则使用普通合并函数,这时时间复杂度和快速排序一个数量级,都是 $O(n\log_2 n)$;如果没有额外空间,则使用关键函数 merge_without_buffer 就地合并,这个合并过程不需要额外的存储空间,但时间复杂度变成 $O(n\log_2 n)$,这种情况下,总的 stable_sort 时间复杂度是 $O(n\log_2 n)$。

部分排序功能能够完成一段数据(而不是所有)的排序,在适当的场合使用可以减少计算量。

partial_sort 和 partial_sort_copy 两个函数能够将整个区间中给定数目的元素进行排序。也就是说,结果中只有最小的 M 个元素是有序的。partial_sort 接受 3 个参数,分别是区间的开头、中间和结尾。执行后,将从中间到开头的 M 个元素有序地放在前面,从中间到结尾的元素肯定比前面的元素大,但它们内部的次序没有保证。partial_sort_copy 的区别在于把结果放到另外指定的迭代器区间中。下面是一个具体的实例。

```
void func()
{
    int ar[12]={69,23,80,42,17,15,26,51,19,12,35,8};
    //只排序前 7 个数据
  partial_sort(ar, ar+7, ar+12);
    //结果是 8 12 15 17 19 23 26 80 69 51 42 35,后 5 个数据不定
  vector<int> res(7);
    //前 7 项排序后放入 res
    partial_sort_copy(ar, ar+7, res.begin(), res.end(), greater<int>());
}
```

这两个函数的实现使用的都是堆结构,先将前 M 个元素构造成堆,然后顺序检查后面的元素,看是否小于堆的最大值,若是,则彼此交换,并重排堆;最后将前面已经是最小的 M 个元素构成的堆作一次 sort_heap 即完成。算法的时间复杂度约为 $O(n\log_2 M)$。

nth_element 函数只真正排序出一个元素,即第 n 个。函数有 3 个迭代器的输入(当然,还可以加上一个谓词),执行完毕后,中间位置指向的元素保证和完全排序后这个位置的元素一致,前面区间的元素都小于等于后面区间的元素。STL 中基本保证其平均时间复杂度为线性阶。

STL 中还有许多其他的排序函数,如 heap_sort、is_sorted_until、q_sort 等,感兴趣的读者可以参考标准库说明文件。

9.9 综合案例——比赛排名问题

1. 问题描述

N 支甲级足球队某年进行了角逐比赛,精英们两两交手,比的是谁犯错误少,假设战绩表只记录输球数,不计净胜球,请根据比赛的数据模拟比赛过程,绘制相应的战绩表,依照水平由高到低对 N 个队的竞赛结果进行排名,并输出本次赛事的冠军队伍。

2. 解题思路

胜者树和败者树都是完全二叉树,是树形选择排序的一种变形。每个叶子结点相当于

一个选手，每个中间结点相当于一场比赛，每一层相当于一轮比赛。败者树是胜者树的一种变体。在败者树中，用父结点记录其左、右子结点进行比赛的败者，而让胜者参加下一轮比赛。败者树的根结点记录的是败者，需要加一个结点记录整个比赛的胜利者。败者树可以简化重构的过程。图 9.15 是一棵败者树，规定**数大者败**。

step1：b3 对决 b4，b3 胜 b4 负，内部结点 ls[4]的值为 4。

step2：b3 对决 b0，b3 胜 b0 负，内部结点 ls[2]的值为 0。

step3：b1 对决 b2，b1 胜 b2 负，内部结点 ls[3]的值为 2。

step4：b3 对决 b1，b3 胜 b1 负，内部结点 ls[1]的值为 1。

注意：在根结点 ls[1]上又加了一个结点 ls[0]=3，记录最后的胜者。

败者树的重构过程为：将新进入选择树的结点与其父结点进行比赛，将败者存放在父结点中；而胜者再与上一级的父结点比较。比赛沿着到根结点的路径不断进行，直到 ls[1]处。把败者存放在结点 ls[1]中，胜者存放在 ls[0]中。图 9.16 是当 b3 变为 13 时，败者树的重构图。虚线框表示一轮比较中的胜者，阴影框表示变动的结点，新加入结点底部以下画线标识。

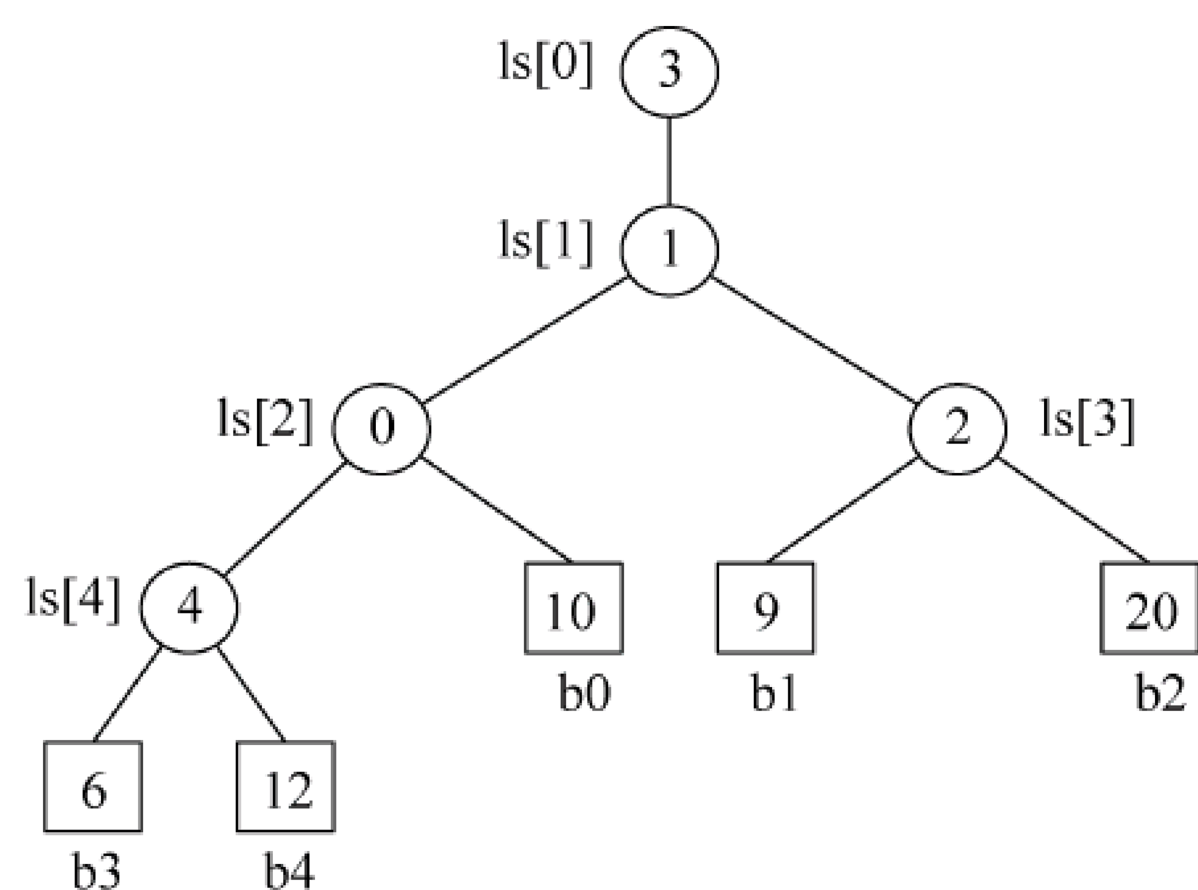


图 9.15　败者树

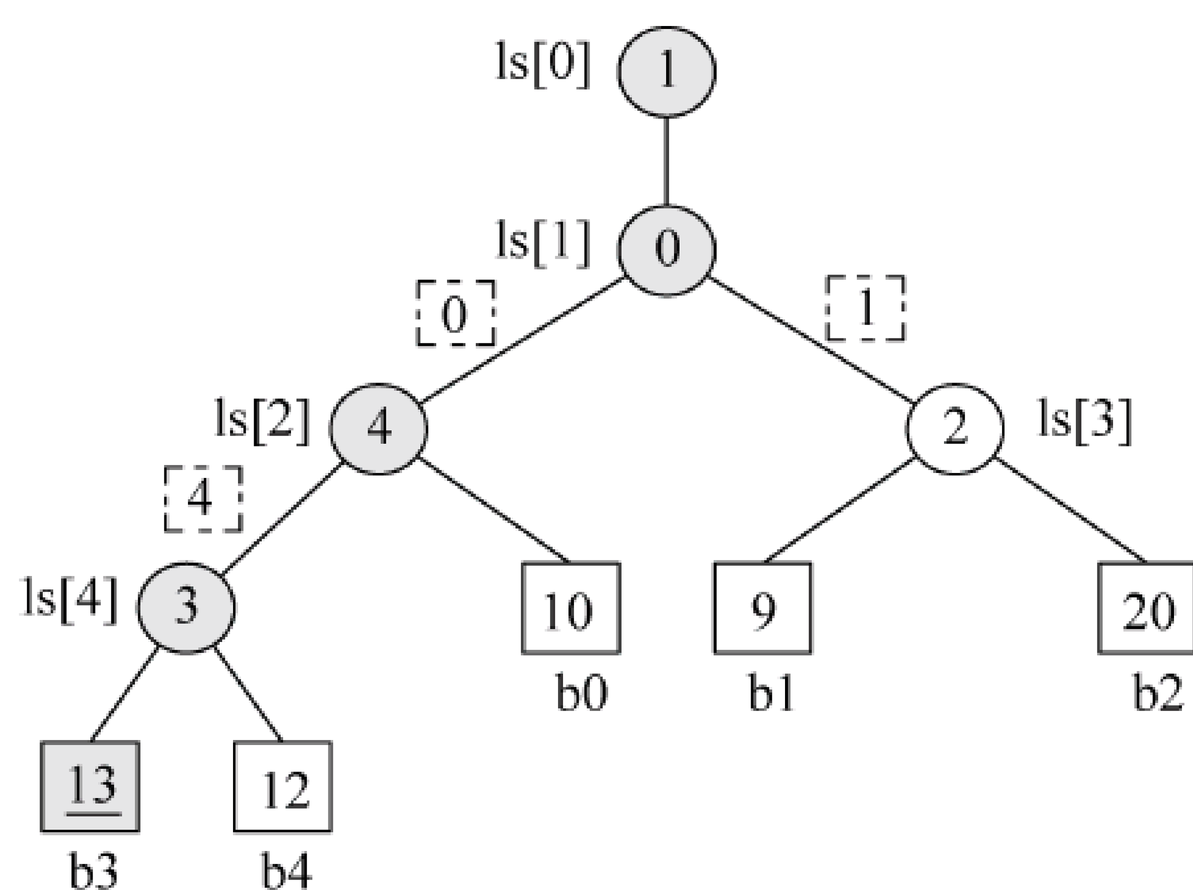


图 9.16　败者树的重构图

注意：败者树的重构只需要与其父结点比较。由图 9.16 可见，b3 与结点 ls[4]的原值比较，ls[4]中存放的原值是结点 4，即 b3 与 b4 比较，b3 负 b4 胜，则修改 ls[4]的值为结点 3。同理，以此类推，沿着根结点不断比赛，直至结束。

3. 数据表示

败者树常常用于多路外部排序，对于 K 个已经排好序的文件，将其归并为一个有序文件。这一特性恰好和 N 支参赛队伍进行对决排名相吻合。败者树的叶子结点是数据结点，即记录的是比赛中队伍的犯错次数，两两分组，内部结点记录左、右子树中的“败者”，优胜者往上传递，一直到根结点，如果规定优胜者是两个数中的较小者，则根结点记录的是最后一次比较中的败者，也就是第二小的数，而用一个变量记录最小的数。

注意：当叶子结点的个数变动时，需要完全重新构建整棵树。

4. 代码实现

败者树的构造：先构造一棵空的败者树，然后把叶子结点逐一插入败者树，自底向上不断调整，保持内部结点保存的都是失败者的结点编号，优胜者一直向上不断比较，最终得到一棵合格的败者树。

```
//叶子结点的个数为 K
#define K 100
//从下标 1 开始存储叶子结点值,下标 0 处存储一个最小值结点,用来初始化败者树
int leaves[K+1];
//冠军结点存储在下标 0,下标 1 到 K-1 存储内部结点,即存储中间结点值
int loserTree[K];
void adjust(int i) {
  int parent = (i+K-1)/2;       //求出父结点的下标
  while(parent > 0) {
    if(leaves[i] > leaves[loserTree[parent]]) {
      int temp = loserTree[parent];
      loserTree[parent] = i;
      i = temp;                 //i 指向的是优胜者
    }
    parent = parent / 2;
  }
  loserTree[0] = i;
}
void initLoserTree() {
  int i;
  for(i = 1; i < K + 1; i++)
    scanf("%d", &leaves[i]);
  leaves[0] = MIN;
  for(int i = 0; i < K; i++)
    loserTree[i] = 0;
  for(int i = K; i > 0; i--)
    adjust(i);
}
```

本章小结

本章主要讲解了几种常见的内排序方法,主要学习要点如下。

- 理解排序相关的概念和分类。
- 掌握插入排序的基本思想和实现过程,分析并总结插入排序的效率。
- 掌握交换排序的基本思想和实现过程,分析并总结交换排序的效率。
- 掌握选择排序的基本思想和实现过程,分析并总结选择排序的效率。
- 掌握归并排序的基本思想和实现过程,分析并总结交换排序的效率。
- 了解基数排序的基本思想和实现过程。
- 理解排序方法在实践中的应用。

第 10 章 ACM 经典案例

10.1 递 归 算 法

- 定义是递归的。
- 数据是递归的。
- 解决问题的过程是递归的。

10.1.1 三柱汉诺塔问题

1. 问题描述

汉诺塔(Hanoi)问题是一个古典的数学问题,它是一个用递归方法求解的典型例子。古代有一个梵塔,塔上有 3 个底座 A,B,C,开始时 A 座上有 64 个盘子,盘子的大小相等,规定摆放时,大的在下,小的在上,且每一次只能移动一个盘子。有一位老和尚想把这 64 个盘子从 A 座移到 C 座,每次只允许移动一个盘子,且在移动过程中盘子均在 3 个座上,又始终保持大盘在下,小盘在上。图 10.1 为汉诺塔模型。

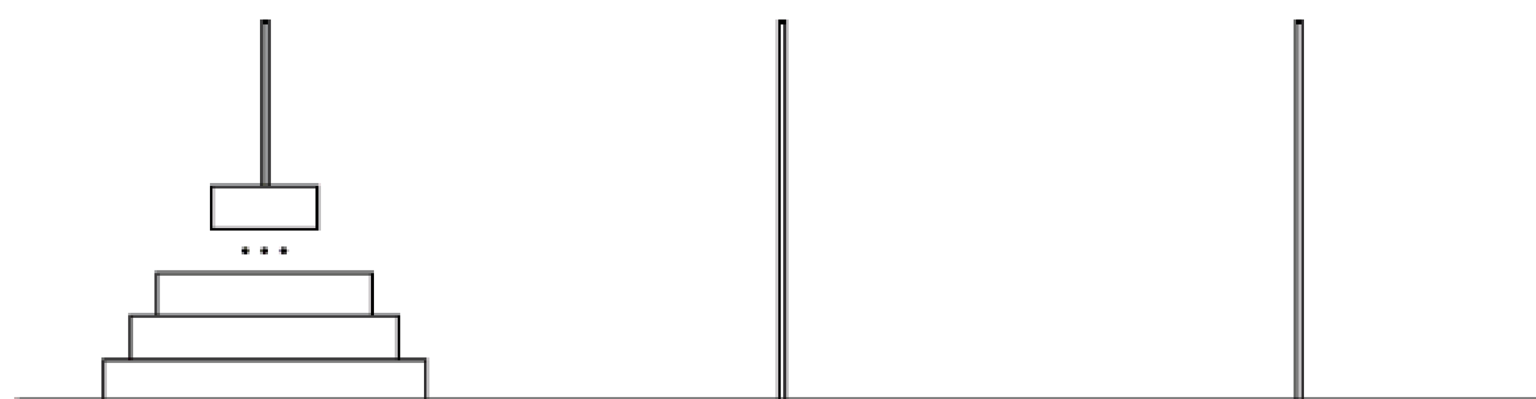


图 10.1 汉诺塔模型

2. 解题思路

下面以 3 阶汉诺塔的移动为例,分析这个问题应该如何求解。图 10.2 是 3 阶汉诺塔的移动过程。若想移动最底部的 3 号盘子,则必须移开它上面的 2 号盘子,而若想移动 2 号盘子,又必须移动上面的 1 号盘子。所以,步骤(1)将 1 号盘子从 A 移到 C,再经过步骤(2)将 2 号盘子从 A 移到 B。这时再经过步骤(3)将 1 号盘子从 C 移到 B,并经过步骤(4)将 3 号盘子从 A 移到 C。这样,3 号盘子就移动到 C 上了。再考虑将 2 号盘子移动到 C 上。所以,经过步骤(5)将 1 号盘子从 B 移到 A,再经过步骤(6)将 2 号盘子从 B 移到 C,最后经过步骤(7)将 1 号盘子从 A 移到 C,至此整个过程就完成了。

当盘子数增加时,只要按着这样的递归规则移动,最初的大问题就会逐个分解为规模更小的问题,求解难度也随之降低。推广开来,当盘子的数目为 1 时,只要将盘子从塔座 A 直

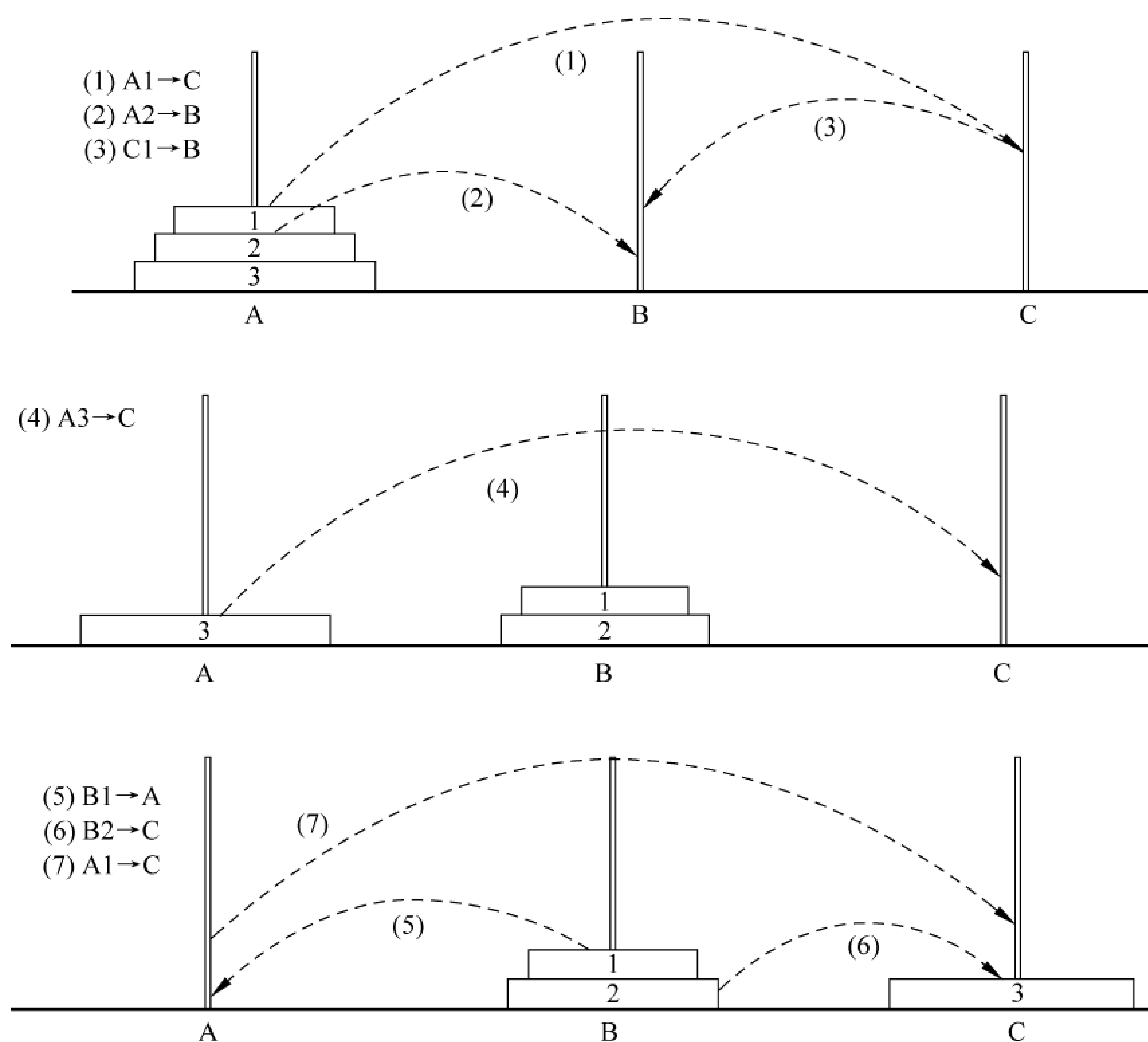


图 10.2　3 阶汉诺塔的移动过程

接移动到 C 上即可。当盘数大于 2 时,则需要利用塔座 C 辅助中转,而 C 作为临时目标塔座。这时须想办法将 n－1 个较小的圆盘依照规则从 A(经过 C)移动到 B 上;再将剩下的最大的盘子从 A 移动到 C;最后,再将 n－1 个小盘依照规则从 B(经过 A)移动到 C。如此递归进行下去,n 个圆盘的移动问题就可以分解为**两次** n－1 个圆盘加上**一次** 1 个圆盘的移动问题,也就是**分而治之**。

3. 代码实现

```
#include <iostream>
using namespace std;
void move(int n, char x, char y) {
  cout << "Move " << n << " from " << x << " to " << y << endl;
}
void hanoi(int n, char a, char b, char c) {
  if (n == 1)
    move(1, a, c);
  else {
    hanoi(n-1, a, c, b);
    move(n, a, c);                //单次直接移动
    hanoi(n-1, b, a, c);
```

```
    }
  }
  int main(int argc, char * * argv) {
    int num;
    cout << "输入盘子数: " << endl;
    cin >> num;
    cout << "汉诺塔的解法,盘子数为 " << num << endl;
    hanoi(num, 'a', 'b', 'c');
    return 0;
  }
```

10.1.2 传染病问题

传染病流行学作为一门新兴的学科,致力于研究对传染病发展传播起到相当程度作用的各种因素之间的相互关系。该领域的一个研究焦点就是研究在单一有机体内传染病发展的影响因子模型,进而对整个人群的传染病发展的影响因子模型进行研究。

1. 问题描述

现在需要完成一个用以评估某组织样本中感染水平的程序。输入的数据是一个矩形的组织样本,为实现数字化,方便计算机处理,将这个矩形的组织样本用一个由 0 和 1 组成的二维阵列表示,组织中的某一部分可能被感染,也可能没有被感染。正如图 10.3 所示的那样,当组织中的一个细胞被感染后,就用 1 标识;相反,一个健康的细胞用 0 标识。

1*	0	0	0	0	0
0	1*	1*	1*	0	0
0	0	1*	1*	0	1
0	0	1*	0	0	1
1	0	0	0	1	1
1	0	0	0	0	0

图 10.3 感染的菌群示意图

图 10.3 中存在 3 个**菌群**。最小的菌群位于左下角,仅由两个细胞组成。另外一个稍大一点的菌群位于图像的右侧,由 4 个细胞组成。最大的菌群由 7 个被感染的细胞组成。为了估算机体受感染的程度,该程序需要在接收一个坐标为输入后,以该点为中心向周围的 8 个方向递归扩展,并检查周围的细胞是否被感染。假设当输入为(1,1)时,得到的菌群用“1 * ”表示。

2. 解题思路

这同样是一个运用**分治法**进行解决的典型问题。与汉诺塔问题的区别在于,它的**子问题数**不再是两个,而是 8 个,因为每个点都有 8 个方向,有 8 个临近点(除了边缘点以外)。另外,这个问题在数字图像处理中也会遇到。它可以被看作是**种子算法**的一种变形,二者都是从搜索的焦点(单点)向四周扩散,就像是先播下的种子随着时间不断生长、成熟,并朝着四周传播孕育新生命的种子。

需要着重说明的是,这个问题的**关键**在于运用递归向四周发散式搜索时,一定要注意排除已经检查过的点(本实例使用辅助数组 visited[ ][ ]标识已探查的点),否则程序将陷入死循环。另外,同样需要注意边界的检查,以避免越界操作。下面给出了该问题的求解示例程序。

3. 代码实现

```
#include <iostream>
#include <string>
#include <cstdlib>
using namespace std;

const int ROWS = 6;
const int COLS = 6;
const int TOTAL = 13;

const int maxn = 100 + 5;
char map[maxn][maxn];
char visited[maxn][maxn];

typedef struct {
    int x, y;
} point;
point graph[TOTAL] = {                                          //测试数据
    {0,0}, {1,1}, {1,2}, {1,3}, {2,2}, {2,3}, {3,2},            //set1
    {2,5}, {3,5}, {4,5}, {4,4},                                 //set2
    {4,0}, {5,0}                                                //set3
};

//在坐标(r,c)指示的细胞的所在菌落内部计算被感染细胞总数
int calc(char map[][maxn], int r, int c);
void output(char map[][maxn], int r, int c);

int main(int argc, char * * argv) {
    //初始化
    for (int i = 0; i < ROWS; ++i)
        for (int j = 0; j < COLS; ++j)
            map[i][j] = '0';

    //填入菌落
    for (int i = 0; i < TOTAL; ++i) {
        int x, y;
        x = graph[i].x;
        y = graph[i].y;
        map[x][y] = '1';
    }

    int col, row;
    cout << "请输入要检测的坐标(格式 x,y)" << endl;
    cin >> row >> col;

    output(map, row, col);
    cout << "包含细胞 (" << row << ", " << col << ") 的菌群共有"
```

```
        << calc(map, row, col) << "个感染细胞."
        << endl;
    output(map, row, col);

    return 0;
}
int calc(char map[][maxn], int r, int c) {
    if (r < 0 || c < 0 || r >= ROWS || c >= COLS || visited[r][c])
        return 0;
    int sum = 0;
    if (map[r][c] == '1') {
        visited[r][c] = true;
        //由单点向四周(8个方向)进行搜索
        for (int i = -1; i <= 1 ; i++)
            for (int j = -1; j <= 1; j++)
                if (i == 0 && j == 0) sum++;
                else sum += calc(map, r+i, c+j);
    }
    return sum;
}

void output(char map[][maxn], int r, int c) {
    for (int i = 0; i < ROWS; ++i) {
        for (int j = 0; j < COLS; ++j) {
            cout << map[i][j];
            if (visited[i][j])
                cout << "* ";
            else
                cout << " ";
        }
        cout << endl;
    }
    cout << endl;
}
```

10.1.3 N皇后问题

回溯法也称试探法，它是一种系统地搜索问题的解的方法。“**回溯**”这个词本身就有逆流而上的意思。回溯的过程正是当某一种可能的试探结果**否定了该可能路径的正确性**后，**回退至先前的某个状态**继续进行**其他可能性的试探**的过程。其中，否定某个可能路径很像果园的果农将不会长出果实的枯枝丫剪掉的“剪枝过程”，将算力资源调度到别处需要的地方，从而达到节省计算资源(果树养分)，加快全体解空间的搜索(果树的开花结果)效果。可以说，回溯策略并不是按照某种固定的计算方法计算的算法，而是通过尝试探索和纠正错误寻找答案。

1. 问题描述

本小节要解决的N皇后问题便是回溯算法的典型例题，该问题是19世纪德国著名数

学家高斯在1850年提出的,问题的要求是在N行N列的国际象棋棋盘上摆放N个皇后的同时,满足它们之间互相不处于各自的攻击范围,即互不攻击。那么,皇后的攻击范围该如何判定呢?在国际象棋中,皇后是最强大的棋子,它的攻击范围最大,若两个皇后位于**同一行**、**同一列**或**同一对角线**上(注意区分主副对角线),则称它们可以**互相攻击**。图10.4中的阴影部分直观地展示了皇后的攻击域。

需要解释的是,下面的代码是通用的,适用于任何合法的正数值(应不超过预设的棋盘最大宽度)。当然,棋盘的空间越大,相应的有效解越多,要全面而不遗漏地求出这些解(合法的皇后摆放位置),自然会耗费更多的CPU算力。为方便下文的算法讲解和插图绘制,本书默认取N为4,感兴趣的读者可自行运行代码,尝试输入不同的棋盘阶数检验自己的理解。

2. 解题思路

现在来看如何使用回溯法解决N皇后的问题。这个算法将在棋盘上的无冲突地摆放皇后,直到N个皇后在不相互攻击的情况下都被摆放在棋盘上。

由图10.5可知,一个合适的解应当在每行、每列上只有一个皇后,且在一条斜线(斜对角线)上也只有一个皇后。

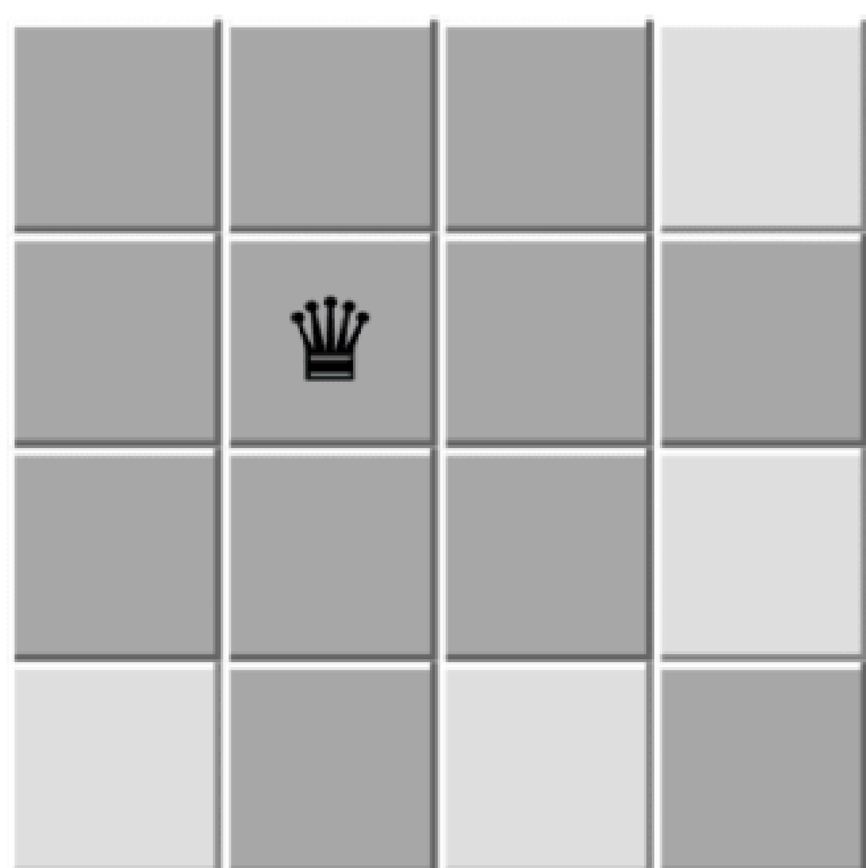


图10.4 皇后的攻击域

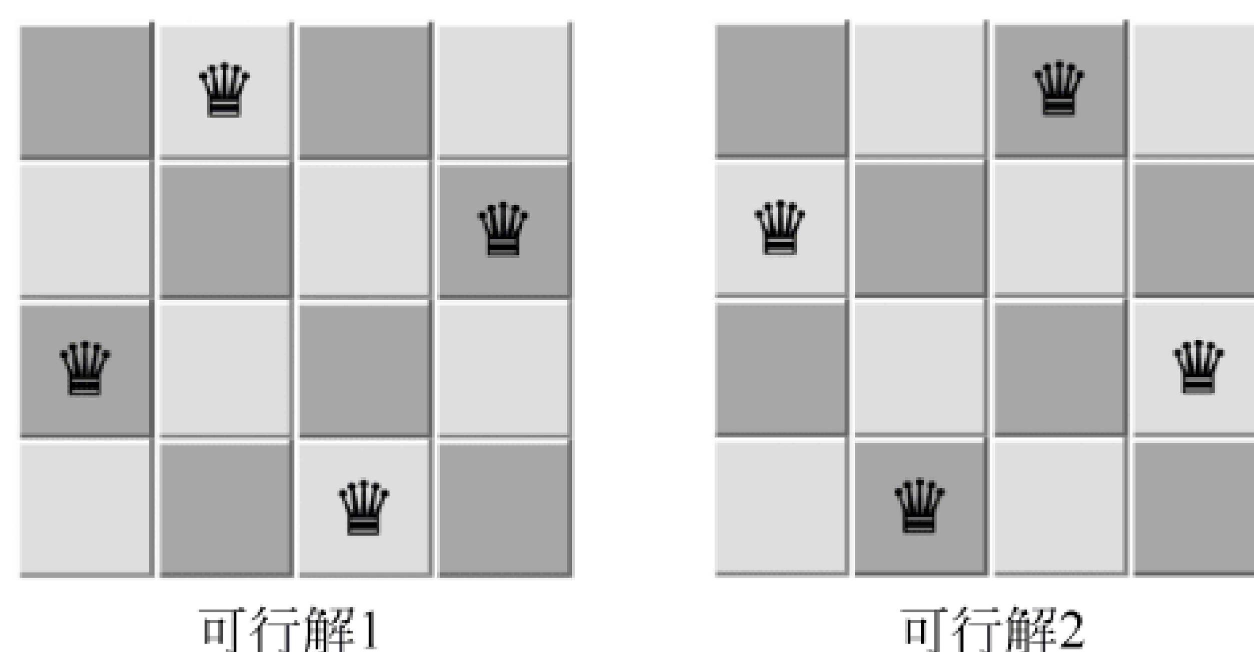


图10.5 4阶棋盘的不相互攻击的两种情况

求解过程从空棋盘(视作是n×n的方阵)开始。假设在第1~m行均为合理放置方案的基础上,接着再去处理第m+1行,直到第n行已经完成放置时,则表示已经求得一个合法解。接着回退至上一层递归(正在处理n−1行棋盘)实例,去改变第n行的放置位置(即切换到另一还未访问的列),期望能够获得下一个解。**相反**,在另一种情形下,若是第i行无法合法地放置第i位皇后,即第i个皇后放在第i行的任一个列位置都会处于其他皇后的攻击范围,那么就放弃这一条分支,不再继续向下搜索,而是向后**回溯**,去改变第i−1行的放置方案。

由上述过程可知,关键步骤是搜索。下一步需要考虑的是如何从庞大的解空间(原始规模为n^n)**全面彻底**且**不重复**地搜索出我们想要的解,与此同时,要保证算法**足够高效**,即如果对当前的搜索路径继续搜索下去不可能构成一个合法的解时,必须将该分支**"剪掉"**,不再向下探索。接着再考虑极端情况,若当前行的每一列均不可放置皇后或探索过了,那么下面会无路可走,必须**回溯**至上一行,重新选择上一行的另一未经探索的列号继续搜索。

在下面算法搜索有效解的过程中,我们是以行为处理单位,**先**从低到高,逐一递归地处

理每一行,**再**进入每一行的内部,自左而右,逐一遍历每一列去放置皇后。每一个递归实例中的皇后都不在同一行上,绝对不可能互相横向攻击。因此,我们只需要考虑剩下的 **3 种可能**的相互攻击的情况。

- 同一列的皇后相互攻击。
- 同一条主对角线的皇后互相攻击。
- 同一条副对角线的皇后互相攻击。

针对第一条,设计用于存储**皇后放置方案**的空间安排见下面的**存储结构**。

由图 10.6 可见,只通过比较解数组中的**内容**(值),便可判断任意两个皇后是否处于同一行。对于后两种相互攻击的情况,借助图 10.7 中的两张取值表,不难发现它们之间的规律——对于棋盘上的任意两点 x,y,有

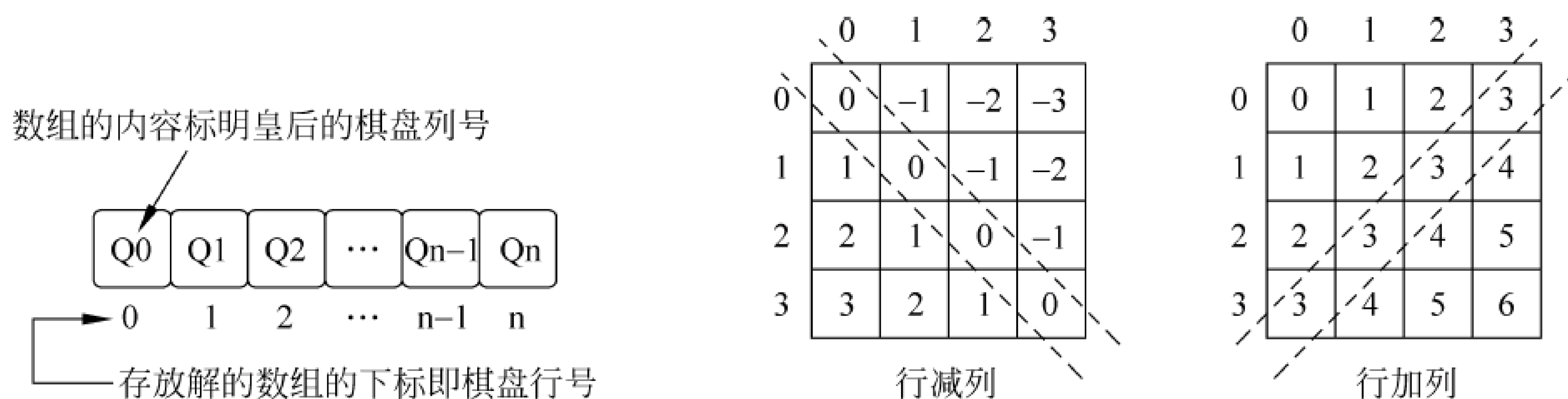


图 10.6 解数组的存储结构说明　　图 10.7 处于同一条(主/副)对角线上元素的空间分布特征

它们的行下标分别减去它们的列下标 x. row－x. col＝＝y. row－y. col,若相等,则两个点处于相同的**主对角线**上。

它们的行下标分别加上它们的列下标 x. row＋x. col＝＝y. row＋y. col,若相等,则两个点处于相同的**副对角线**上。

到此为止,棋盘上皇后每一种可能相互攻击的情况都已经覆盖到了,这些不合法的情况将用于判断语句中,用来避免一些不必要的求解过程,即将**不能**"生长出"我们所需要的解的枝丫剪掉。下面给出相应的代码。

3. 代码实现

```
#include <cstdio>
#include <cmath>
#include <cstring>
#include <ctime>

/* Cpp library */
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;
const int MaxWidth = 20 + 1;   //棋盘的最大宽度(方阵的最大阶数)
const int MaxSize = 1000 + 10;
```

```
//理想正解，列数组，棋盘相关统计变量及存放数组
int ideal_tot, Col[MaxSize], N, Solution_Pos[MaxWidth];

//各模块声明,分别是输入辅助函数、可行解棋盘图形绘制函数、可行解搜索函数
int input(int &n);
void draw_solution();
void search(int row);

void draw_solution() {
  cout << "Solution_Pos #" << ideal_tot << ": \n";

  string hashBarrier(N * 3+9, '#');
  string lineBarrier(N * 3+9, '-');
  lineBarrier[0] = '#';
  lineBarrier[6] = '+';
  lineBarrier[lineBarrier.size()-1] = '#';

  cout << "\n" << hashBarrier << endl;
  cout << setw(3) << "# r/c |";
  int r, c;
  for(c = 0; c < N; ++c)
    cout << setw(3) << c;
  cout << " #" << endl;
  cout << lineBarrier << endl;

  for(r = 0; r < N; ++r) {
    cout << "#" << setw(3) << r << " |";
    for (c = 0; c < N; ++c) {
      if(Solution_Pos[r] == c)
        cout << setw(3) << "Q";
      else
        cout << setw(3) << "0";
    }
    cout << " #" << endl;
  }
  cout << hashBarrier << endl;
}
//皇后可攻击域(同列、同主对角线、同副对角线)
//这里解释一下,本算法是按行放置皇后的,故绝对不会横向攻击
void search(int row) {
  if(row == N) {
    ideal_tot++;
    draw_solution();
    return;
  }

  //遍历当前行的皇后欲放置的列 from i to n
  for (int cur = 0; cur < N; ++cur) {
```

```
        //假设冲突不存在
        bool bConflict = false;
        Col[row] = Solution_Pos[row] = cur;
        //与之前行的皇后,检测其位置是否有互相攻击的冲突
        for(int pre = 0; pre < row; ++pre) {
            if ( Col[row] == Col[pre]               ||           //检测同列
                (row - Col[row]) == (pre - Col[pre]) ||          //检测主对角线
                (row + Col[row]) == (pre + Col[pre]) ){          //检测副对角线
                bConflict = true;
                break;
            }
        }
        if (!bConflict) search_ease(row + 1);
    }
}

int input(int &n) {
    printf("请输入棋盘阶数:\n> ");
    return scanf("%d", &n);
}

int main(int argc, char * * args) {
    time_t tStart, tEnd;

    while (input(N) == 1 && N != EOF) {
        if (!(N >= 1 && N < MaxWidth)) break;
        ideal_tot = 0;
        memset(Col, 0, sizeof(Col));
        memset(Solution_Pos, -1, sizeof(Solution_Pos));

        cout << N << " 阶" << "棋盘:" << "试探回溯法进行时" << endl;
        time(&tStart);
        search(0);
        time(&tEnd);
        cout << "理论上全部可行解共有: " << ideal_tot << " 个\n"
            << "搜索过程所耗费的时间: " << tEnd - tStart << " 秒\n"
            << "程序整体用时(毫秒): " << (double) clock() / CLOCKS_PER_SEC
            << endl;
    }
    return 0;
}
```

10.2 DFS 与 BFS 问题

深度优先搜索(Depth First Search,DFS)遵循的搜索策略是尽可能“深入”地搜索图顶点。在深度优先搜索中,对于最新发现的顶点,如果它还有以此为起点还未探测到的边,就沿着此边继续搜索下去。当结点 v 的所有点都已经被探寻过,搜索将回溯到射入本顶点的

那条弧所连接的起始顶点。这一过程一直进行到从源结点可到达的所有结点都访问到为止。如果此时还存在未被访问的结点(即这张图并不是任意两个结点都是可连通的连通图),则选择未访问顶点中的一个作为源结点并重复以上搜索过程,整个过程反复进行,直到所有结点都被访问为止。

广度优先搜索(Breadth First Search,BFS)和 DFS 类似,只不过 BFS 每次都先尽可能扩展当前顶点的邻接结点(即搜索策略的着重点是尽可能地"**广博**"),之后再向其子结点进行扩展,因此需要一个先进先出(First In First Out,FIFO)队列暂时存放当前结点的邻接顶点。扩展的过程和二叉树的**层序遍历**十分相像。广度优先搜索的操作步骤如下,具体流程如图 10.8 所示。

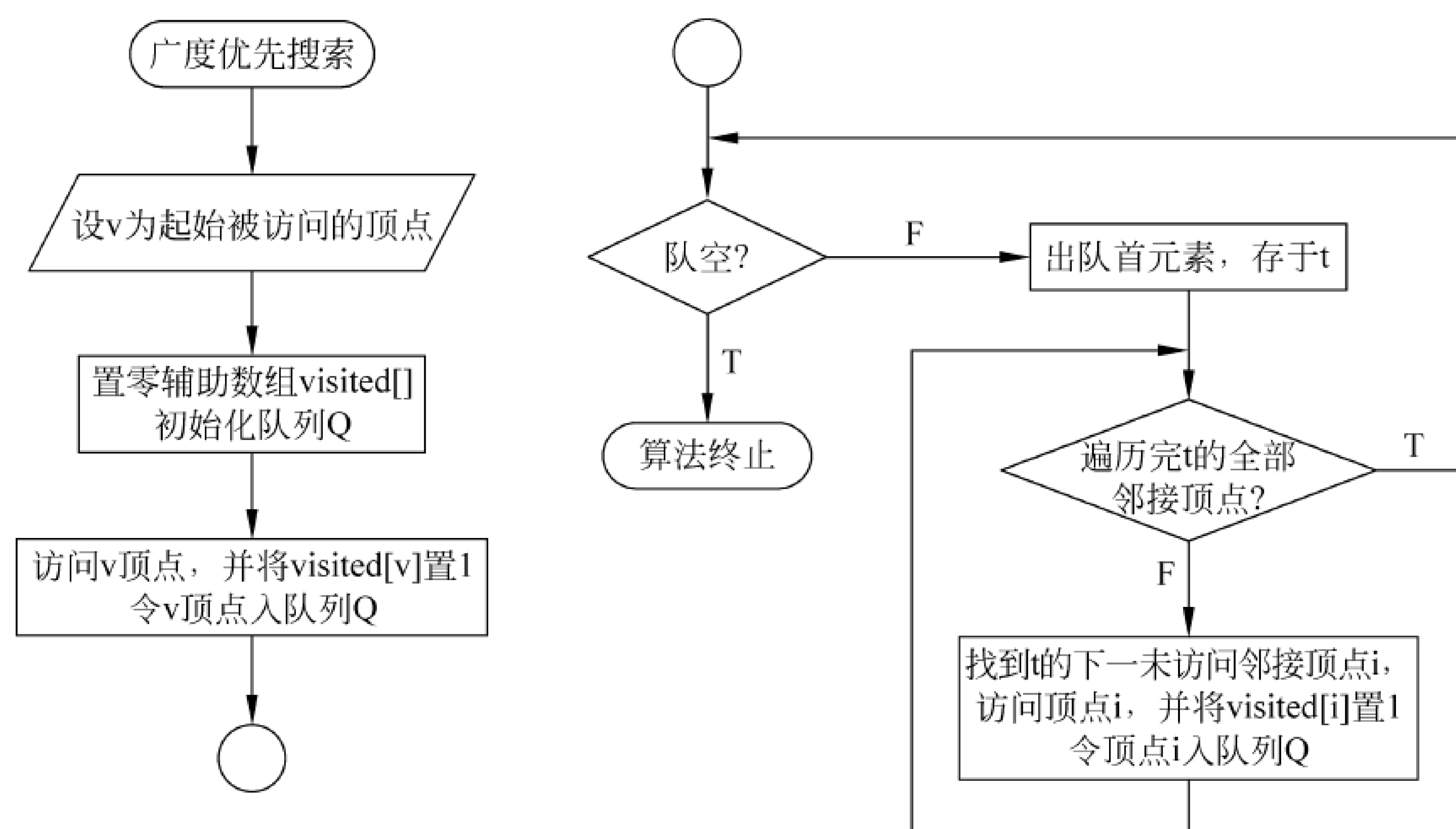


图 10.8　广度优先遍历算法流程图

(1) 从结点 v 开始,给 v 标上已到达(或已访问)的标志——此时称结点 v 还没有被检测。当算法访问了邻接于该结点的所有结点时,称该结点被检测了。

(2) 访问邻接于 v 且尚未被访问的所有结点——这些结点是新的未被检测的结点。将这些结点依次放置到一个未检测结点队列(队列 Q)中(从末端插入)。

(3) 标记 v 已经被检测。

(4) 若队列 Q 为空,则算法终止;否则,从队列 Q 的队头取一结点作为下一个检测的结点。

(5) 重复上述过程。

暂时不考虑实际的应用场景,仅以图论的定义为基础,从某一个起点出发,搜索其可到达的所有顶点,基本的代码实现如下。

```
#include <queue>
#include <iostream>
#include "graph.h"                //包含图的存储结构定义
using namespace std;
```

```
//辅助数组,用于记录已经访问的结点,避免重复访问
int visited[MAXV];
void dfs(AGraph * g, int v) {
  ANode * p;

  cout << g->adjlist[v].data << endl;
  visited[v] = 1;
  p = g->adjlist[v].firstarc;
  while (p) {
    if (visited[p->adjvex] != 1)
      dfs(g, p->adjvex);
    p = p->nextarc;
  }
}

void dfs_travel(AGraph * g, int v) {
  int i;
  //初始化辅助数组,表示所有顶点均未被访问过
  //对图中所有未被访问的顶点进行深度优先遍历
  for (i = 0; i < g->n; i++)
    visited[i] = 0;
  for (i = 0; i < g->n; i++)
    if (visited[i] != 1)
      dfs(g, i);
}

//假设处理的图为带权图
void bfs(MGraph * g, int v) {
  queue<int> q;

  cout << g->vex[v].data << endl;
  visited[v] = 1;
  q.push(v);
  while (!q.empty()) {
    int t = q.front(); q.pop();
    for (int i = 0; i < g->n; ++i) {
      //因简单图中不存在环路,所以先排除 0 权值
      //再判断顶点 t 和 i 之间是否有边,约定没有边用权值为"无穷"表示
      if ( (g->edges[t][i] != 0 || g->edges[t][i] != INF) &&
           visited[i] != 1) {
        cout << g->vex[v].data << endl;
        visited[i] = 1;
        q.push(i);
      }
    }
  }
}

void bfs_travel(MGraph * g) {
```

```
    int i;
    for (i = 0; i < g->n; ++i) {
        visited[i] = 0;
    }
    for (i = 0; i < g->n; i++)
        if (visited[i] != 1)
            bfs(g, i);
}
```

graph.h 中声明并定义了图的两种常见的存储结构：邻接表和邻接矩阵，以及一些在算法实现中需要用到的常量，代码如下。

```
#ifndef GRAPH__H
#define GRAPH__H 1
#define INF 32767                   //约定图中不存在的边用 INF(∞无穷)表示
typedef char ElemType;
const int MAXV = 100 + 10;          //图中的最大顶点个数

///邻接表存储结构
typedef struct ArcNode {            //弧的结点结构类型
    int adjvex;                     //该弧的终点位置
    int weight;                     //该弧的权值信息
    struct ArcNode * nextarc;       //指向下一条弧的指针
} ANode;

typedef struct VertexNode {         //邻接表头结点的类型
    elemtype data;                  //顶点信息
    int inDegree;                   //入度信息，仅在拓扑排序中使用
    ANode * firstarc;               //指向第一条弧
} VNode;

typedef struct AdjacentGraph {      //图的邻接表类型
    VNode adjlist[MAXV];            //邻接表
    int n, e;                       //图中的顶点数 n 和边数 e
} AGraph;

///邻接矩阵定义
typedef struct VertexType {         //顶点类型
    int no;                         //顶点编号
    elemtype data;                  //顶点的其他信息，如对应的字母符号
} VType;

typedef struct MatrixGraph {        //图的邻接矩阵类型
    int edges[MAXV][MAXV];          //邻接矩阵
    int n, e;                       //顶点数和弧数
    VType vex[MAXV];                //存放顶点的信息
} MGraph;
#endif
```

10.2.1 DFS 之迷宫难题

1. 问题描述

给定一个 M×N 的迷宫图，求一条从制定入口到出口的路径。假设迷宫图如图 10.9 所示（M=8，N=8）。对于图中的每个方块，空白表示通道，阴影表示墙。所求路径必须是简单路径，即在求得的路径上不能重复出现同一通道方块。

2. 数据组织

为了表示迷宫，并方便用图的邻接矩阵存储，设置一个二维数组表示二阶方阵，因为迷宫是矩形，并不一定总是正方形的，但可以对矩形进行恒等变形。

$$Square_{Length} = Max\{Rect_{width}, Rect_{Length}\}$$

即令方阵的长度为矩形的长和宽中的最大值，再在较短的那条边上采用“补墙”的方法实现从矩形迷宫到方形迷宫的转换。另外，为了算法的方便，在迷宫外围增加了一道宽度为 1 的围墙。图 10.9 所示的迷宫对应的迷宫数组 maze（因迷宫四周加了一道围墙，故 maze 的行数和列数均加 2）如图中的音符♪、♬，分别用于表示起点、终点。

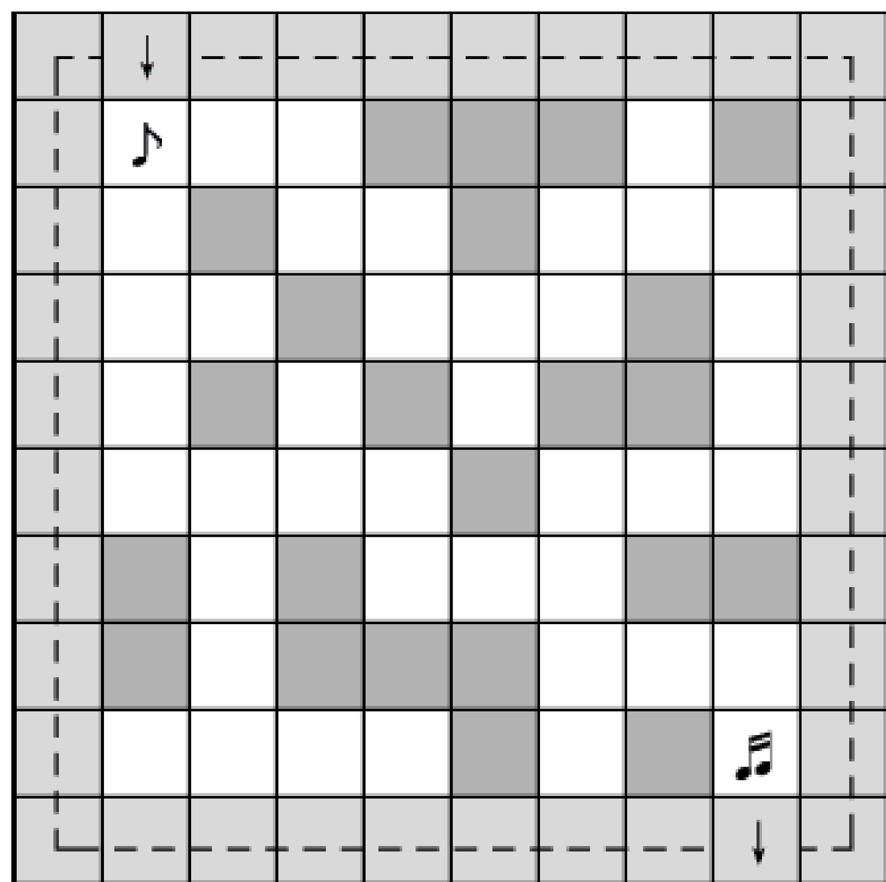


图 10.9 迷宫示意图

```
int maze[M+2][N+2] = {
{1,1,1,1,1,1,1,1,1,1},
{1,0,0,0,1,1,1,0,1,1},
{1,0,1,0,0,1,0,0,0,1},
{1,0,0,1,0,0,0,1,0,1},
{1,0,1,0,1,0,1,1,0,1},
{1,0,0,0,0,1,0,0,0,1},
{1,1,0,1,0,0,0,1,1,1},
{1,1,0,1,1,1,0,0,0,1},
{1,0,0,0,0,1,0,1,0,1},
{1,1,1,1,1,1,1,1,1,1}
};
```

除了需对原来的邻接表存储结构的定义稍做修改（这样才能正确表示二维空间中的顶点坐标）之外，在算法中还必须用到的存储结构如下。方块 Block 用于对迷宫中的每一小段单位路径进行建模，以方阵的左上角作为坐标系原点（0，0），列号从左到右递增，行号自上而下递增。Path 路径的作用一是记录走过的路径，以便后续输出检验，二是为了重蹈覆辙，陷入死循环。

```
typedef struct {
  int r, w;                          //当前方块的行号、列号
} Block;

typedef struct {                     //路径类型定义
```

```
    Block data[MaxSize];                  //路径坐标值
    int length;                           //路径长度
  } PathType;
  //以下为修改后的邻接表类型定义
  typedef struct ANode {                  //边的结点结构类型
    int r, w;                             //该边的终点位置(r,w)
    struct ANode * nextarc;               //指向下一条边的指针
  } ArcNode;
  typedef struct Vnode {                  //邻接表头结点的类型
    ArcNode * firstarc;                   //指向第一条边
  } VNode;
  typedef struct {                        //图的邻接表类型
    VNode adjlist[M+2][N+2];              //邻接表头结点数组(无须记录边和顶点数)
  } ALGraph;
```

现在已经定义了数据的存储结构,可以将二维数组表示的迷宫 maze 转换成邻接表的存储结构。对于迷宫中的每个方块,有上、下、左、右 4 个方块毗邻,如图 10.10 所示。当前方块的第 i 行第 j 列的位置记为(i,j),规定其上方的方位为方位号 0,并按照顺时针递增编号。

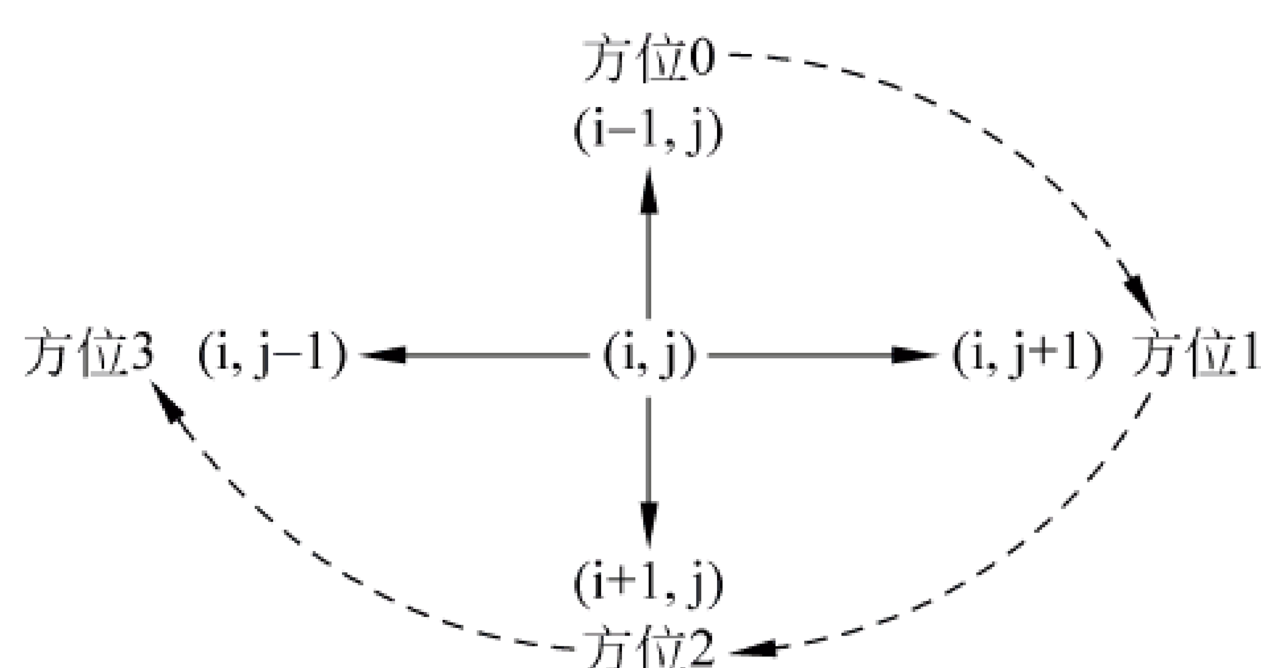


图 10.10　每个方块方位

在转换过程中,假设按照从方位 0 到方位 3 将周围可达的方块依次链入当前方块的邻接链表中。

```
  //基于迷宫数组 maze 转换成等价的邻接表 G
  void InitALGraph(ALGraph * &G,int maze[][N+2]) {
    int i, j, r, c, di;
    G = (ALGraph * )malloc(sizeof(ALGraph));
    ArcNode * p;

    //给邻接表中所有头结点的指针域置初值
    for (i = 0; i < M+2; i++)
      for (j=0; j<N+2; j++)
        G->adjlist[i][j].firstarc = NULL;

    //检查 maze 中的每个元素,更新其邻接链表中的可达邻接点
    for (i=1; i<=M; i++)
```

```
        for (j=1; j<=N; j++) {
            if (maze[i][j] == 1) continue;        //若当前方块为墙,则跳过不处理
            di = 0;
            while (di < 4) {
                //依据顺时针方向,上右下左,链入可达的邻接方块
                switch (di) {
                    case 0: r=i-1; c=j; break;
                    case 1: r=i; c=j+1; break;
                    case 2: r=i+1; c=j; break;
                    case 3: r=i; c=j-1; break;
                }
                //(r,c)为可走向的方块
                if (maze[r][c] == 0) {
                    //创建一个弧结点 * p
                    p = (ArcNode * ) malloc(sizeof(ArcNode));
                    p->r = r; p->c = c;
                    //以首插法将 * p 结点链到链表后
                    p->nextarc = G->adjlist[i][j].firstarc;
                    G->adjlist[i][j].firstarc = p;
                }
                di++;
            }//end_while
        }//end_for
}
```

3. 设计求解程序

求迷宫问题就是在一个错综复杂的图中求从起始到终止顶点的简单路径。在求解时经常采用"穷举求解"方法,即从入口出发,顺着某一方向向前试探,若能走通,则继续向前走,否则沿着原路回溯,换一个方向再继续试探,直至所有可能的通路都已经试探完为止。这恰与深度优先遍历的思想契合。

```
void SearchPath(ALGraph * G, Block now, Block end, PathType path) {
    ArcNode * p;
    //在当前方块的坐标置已访问标记
    visited[now.r][now.c] = 1;
    path.data[path.len].r = now.r; path.data[path.len].c = now.c;
    path.len++;
    if (now.r == end.r && now.c == end.c) {
        printf("\t第%d条可行路径(其长度为%d):\n", ++count, path.len);
        for (int k = 0; k < path.len; k++) {
            printf("<%d,%d>",path.data[k].r, path.data[k].c);
            printf("%c", k == path.len-1 ? '\0' : ' ');
        }
        printf("\n");
    }
    //p 指向顶点 v 的第一条边顶点
```

```
        p = G->adjlist[now.r][now.c].firstarc;
        while (p != NULL) {
            //若(p->r,p->c)方块未访问,则递归访问它
            if (visited[p->r][p->c] == 0)
                SearchPath(G, (Block){p->r,p->c}, end, path);
            //p 指向顶点 v 的下一条边顶点
            p = p->nextarc;
        }
        //当前面的路走不通时,需要原路回溯
        //那么应将当前坐标的已访问标志去掉
        visited[now.r][now.c]=0;
    }
```

4. 主函数调用

编写如下主函数,调用上述算法。

```
#include <cstdio>
#include <cstdlib>
using namespace std;

const int M = 8, N = 8;
const int MaxSize = 100+10;

int main(int argc, char * * argv) {
    ALGraph * G;
    InitALGraph(G, maze);
    PathType path;
    path.len = 0;

    printf("所有的迷宫路径如下:\n");
    DFSearchPath(G, (Block){1,1}, (Block){M,N}, path);

    return 0;
}
```

5. 输出结果

运行图 10.9 所建模的迷宫 maze,求解结果如下。两条迷宫 maze 的可行路径如图 10.11 所示。

所有的迷宫路径如下。

```
第 1 条可行路径(其长度为 15):
<1,1> <2,1> <3,1> <4,1> <5,1> <5,2> <5,3> <5,4> <6,4> <6,5> <6,6>
<7,6> <7,7> <7,8> <8,8>
第 2 条可行路径(其长度为 21):
<1,1> <1,2> <1,3> <2,3> <2,4> <3,4> <3,5> <3,6> <2,6> <2,7> <2,8>
<3,8> <4,8> <5,8> <5,7> <5,6> <6,6> <7,6> <7,7> <7,8> <8,8>
```

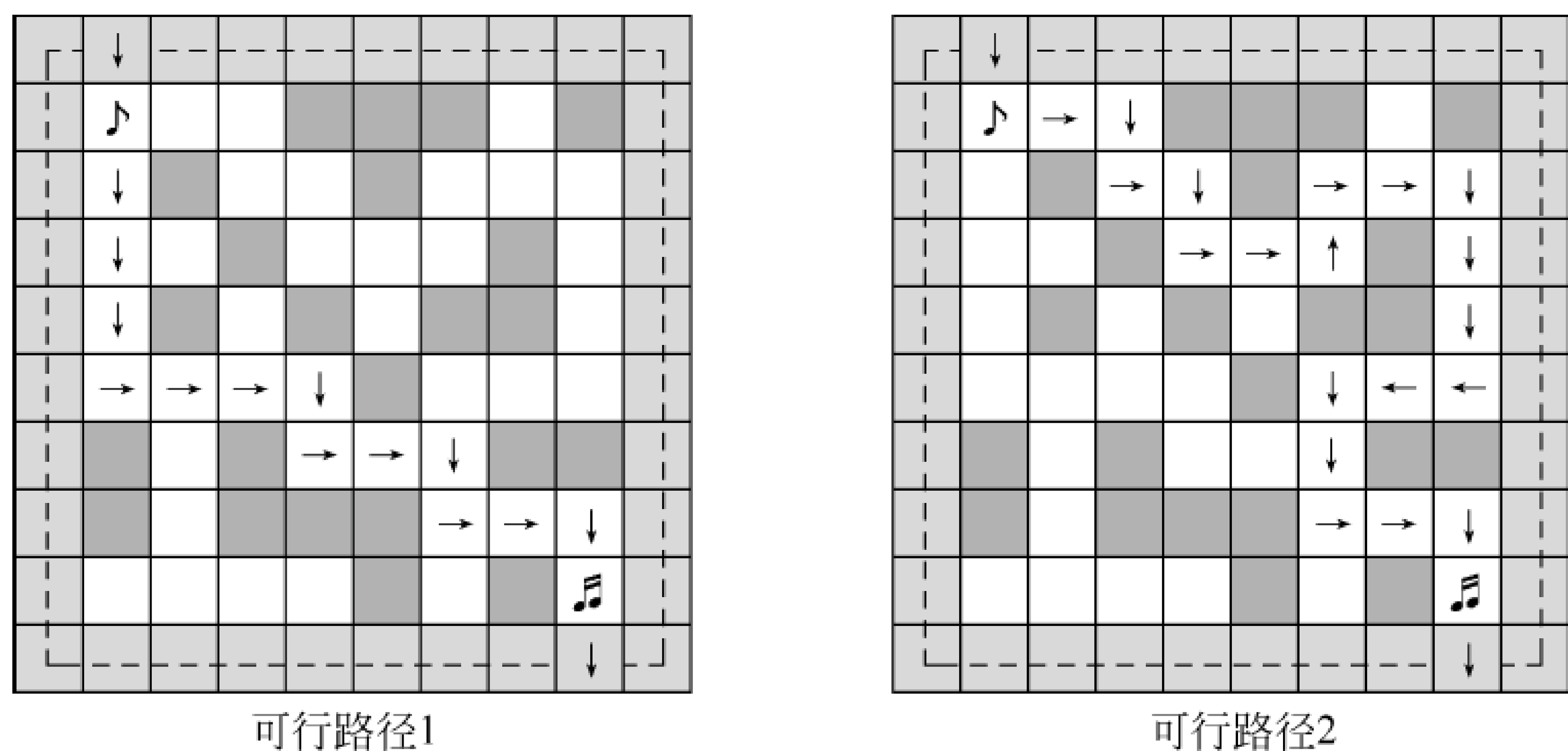


图 10.11　两条迷宫 maze 的可行路径

10.2.2　BFS 之管道和指针游戏

1. 问题描述

管道和指针是一个在 N×N(0<N<20)的方阵上进行的游戏,如图 10.12 所示。将方格从 1～N^2 进行编号,除 1 号和 N^2 号方格以外,其他格子均可能放置管道或指针。管道和指针的数目及其具体的位置由输入决定,它们的数量都在 100 之内,并且管道和指针不能临近放置,也就是在任何放置了两者首尾的方格至今至少还有一个未放置任何东西的方格,并且同一个方格中最多放置一个物品。

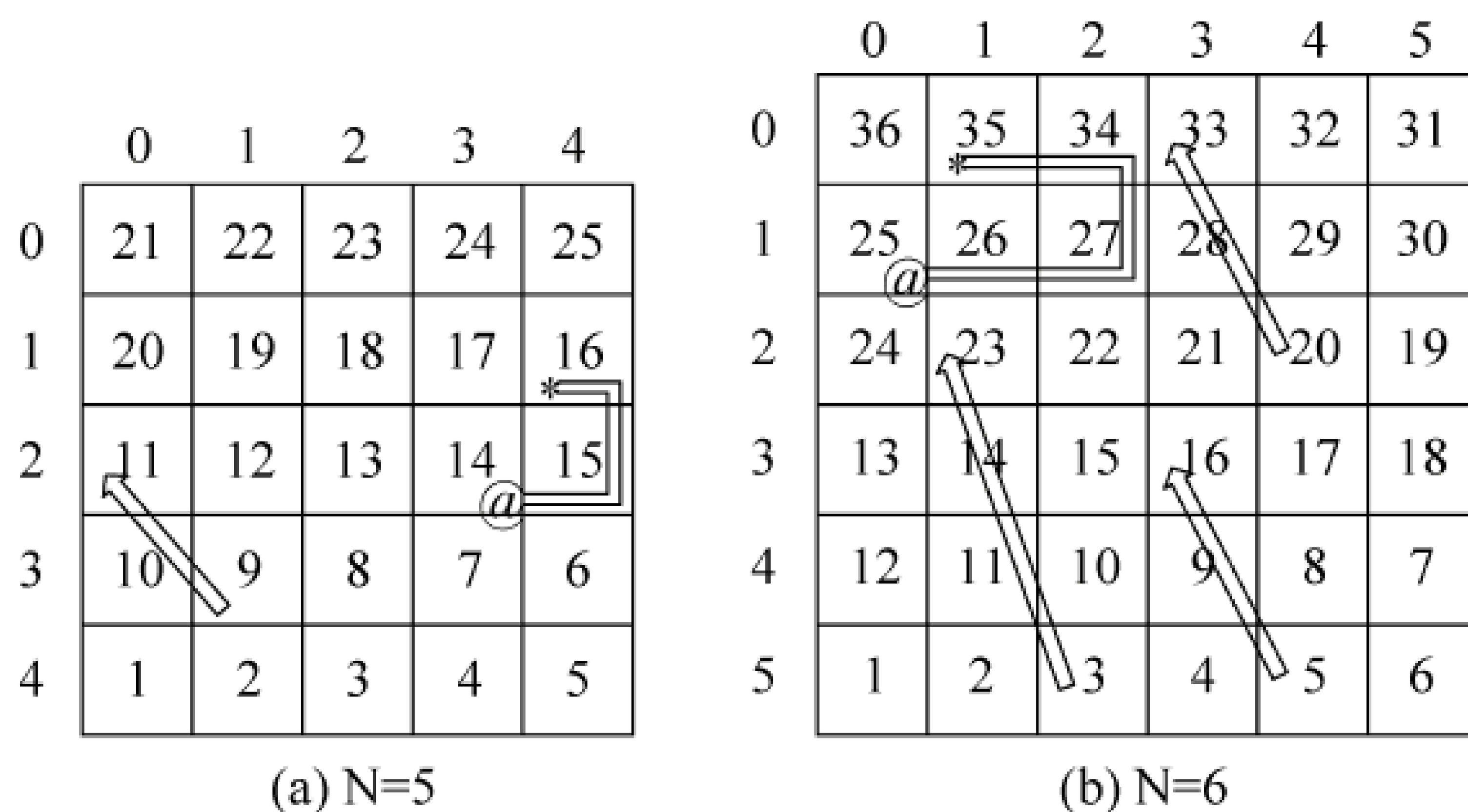


图 10.12　管道和指针

开始时玩家把他们的标志物放在 1 号格子中,玩家轮流以扔骰子的方式移动他们的指示物。如果一个指示物到达了一条管道的入口,则把它移回管道的出口;如果一个指示物到达了一个指针的底部,则将它移动到指针的顶部。

如果你是一个可以自由控制骰子的高手,至少需要扔几次骰子才能到达标号为 N^2 的格子?

step1:**输入**,有多个测试序列。不同的测试数据之间用一个空行隔开,每个测试序列的第一行包含一个正整数 N,表示方阵的大小。如果 N=0,表示输入结束并且不需要处理。**第二行**包含一个整数 M,表示指针的数目。接下来的 M 行,每行包含两个整数 B 和 T,表示指针的底部和顶部。**接下来一行**包含一个整数 K,表示管道的数目。接下来 K 行,每行包含两个整数 O 和 L,表示管道的入口和出口。

step2：输出，每个测试序列输出一行，包含一个正整数，表示题目要求的最少的投掷骰子的次数。

step3：输入样例如下。

```
63
3  23
5  16
20 23
1
35 25
5
1
9  11
1
16 14
0
```

step4：输出样例如下。

```
3
4
```

2. 解题思路

不难看出，这个问题是不能使用贪心算法解决的，因为不能保证在当前这一步所到达的数值比较大的格子，就意味着最优的效率(即使是在这一步中借助指针能到达当前情况下号码最大的格子)。换言之，号码的大小并不能代表从这个格子到达终点的所需要的步数的多少，这就给我们一个启发：管道和指针的本质就是人们经常在游戏中说的“单向传送点”，只不过指针的底部是入口，而顶部是出口。管道的'@'端是入口，而'*'是出口。在计算机中对二者进行建模，完全可以选择相同的结构。

```
struct PipelinePointer {
      int from, to;
} ppnode;
```

接下来要考虑的是解决问题的方法。贪心算法被否定之后，我们的选择可能是搜索，本题采用的搜索应该综合当前这一步能够到达的结点信息去权衡利弊，显然，这是广度优先搜索策略，但是稍加分析之后会发现，如果单纯地采用广度优先搜索，会产生很多重复的结点，现在将指示物所处的某一方格的结点简称为结点 A。

例如，在 6 阶方阵的管道与指针游戏图示中，第一步过后，队列 Q 中存放的结点是 2，23，4，16，6，7，第 2 步时，当结点 2 成为扩展结点时，将入队的结点是"23，4，16，6，8"，其中只有 8 不存在于当前步的结点队列中，即使添加代码加以判断，不把重复的结点再次加入队列中，至少也需要对上一步余留的结点队列进行搜索。

一般而言，采用树状结构和搜索的方法处理问题的关键是利用祖先结点的差异性对孩子结点做不同的处理。在本题中，孩子结点的入队只依赖于父结点的信息，而与其他祖先结

点无关，所以，采用树描述这个过程其实是大材小用。在走了若干步之后，对于一个特定的格子，实际上只有如下两种状态的差别。

- 在走了这些步数之后，存在一种直接的掷骰方案，使指示物位于此格中。
- 不存在一种这样的直接的掷骰移动方案。

因此，可以采用一个 N^2 大小的数组描述若干步之后可以到达的格子的集合(因为集合内部的元素必然互异)，其中每一个元素描述一个格子的状态，0 表示不存在一种方案可到达。1 表示至少存在一种方案可到达。这样，从表示第 n 步状态的数组完全可以推出表示第 n+1 步状态的数组，而且在第 n+1 步的数组得到之后，之前表示第 n 步状态的数组就不再具有使用价值了。一旦数组中表示最后一个格子的元素更新成 1，就表示可以通过这一步完成本问题的求解。

还是上面那个 6 阶游戏案例，描述方阵状态的数组变化过程见表 10.1，为了便于读者阅读，每 N 个值分栏表示，每 N/2 个值插入一个逗号，实际内容不包含这些逗号。

表 10.1 描述方阵状态的数组变化过程

	数组元素的内容(值依次表示从第一格到最后一格的元素的状态)					
描述状态	1～6	7～12	13～18	19～24	25～30	31～36
起始状态	100000	000000	000000	000000	000000	000000
第一步后	010101	100000	000100	000010	000000	000000
第二步后	000101	111111	100111	101111	111110	001000
第三步后	000000	111111	111111	101111	111111	111101

到第三步后，数组的最后一个元素已经变为 1 了，这就表明存在一种投掷骰子的方案，使得扔 3 次骰子，就可以完成任务。以下是实现此算法的主要部分代码，数组下标为 $0 \sim N^2-1$ 的元素分别为第 $1 \sim N^2$ 格的状态，step 用于记录步数，toolbox 是 ppnode 类型的向量，描述了管道和指针代表单向传送门的入口和出口，过程执行完毕后，输出是 step 即可。

另外，此题虽规定方阵为 N^2 大小，但那只是为了输入和举例画图方便。实际上，多大的方阵都可看作是一条直线，使用一维数组足够了。

3. 代码实现

```
int getStep(int n) {
  int i, k;
  int npow = pow(n, 2);
  for (i = 2; i <= npow; i++)
  grid[i] = 0;          //初始化状态数组和步数记录
  grid[1] = 1;
  int step = 0;

  while (grid[npow] == 0) {
    //为当前方格的状态做备份后，归零
    for (i = 0; i <= npow; i++)
      grid_bak[i] = grid[i];
    for (i = 0; i <= npow; i++)
      grid[i] = 0;
```

```
            //搜索所有的格子(最后一个不要搜索,
            //因为在过程结束前它一定一直为零)
            for (i = 0; i <= npow; i++) {
                //若在上一步无法到达此方格,则跳过本轮循环
                if(grid_bak[i] == 0) continue;
                //k 代表投掷骰子的数值
                for (k = 1; k <= 6; k++) {
                    int flag = 0;
                    //若当前所处方格号加上骰子点数大于终点,
                    //则跳出内层 for 循环,因为后续的骰子更大
                    if (i + k > npow) break;
                    for (int l = 0; l < toolbox.size; l++) {
                        //如果骰子指示的方格是一个传送入口,就将传送出口所在的方格标为可达
                        if (toolbox.PP[l].from == i+k) {
                            grid[toolbox.PP[l].to] = 1;
                            flag = 1;
                            break;
                        }
                        //否则,骰子指示的方格指示普通方格,将该方格标为可达
                        if (flag == 0 && grid[i+k] == 0) grid[i+k] = 1;
                    }
                }
            }
            step++;
        }
        return step;
    }
```

4. 主函数调用

```
#include <cstdio>
#include <cmath>
#include <cstdlib>
using namespace std;
//由问题描述知,管道和指针的数量都在 100 之内(即总数不超过 200)
const int MAX = 200+5;

typedef struct PipePointer {
    int from, to;
} ppnode;

typedef struct {
    int size;
    ppnode PP[MAX];
} toolT;

int grid[2 * MAX] = {0}, grid_bak[2 * MAX] = {0}, result[MAX] = {0};
```

```
toolT toolbox;

int main (int argc, char * * argv) {
  //分别对应问题描述的 N, B, K;
  int nGrid, nPtr, nPipe;
  while (scanf("%d", &nGrid) == 1 && nGrid != 0) {
    nPtr = nPipe = 0;
    toolbox.size = 0;
    int i;
    scanf("%d", &nPtr);
    for (i = 0; i < nPtr; i++)
      scanf("%d%d", &toolbox.PP[i].from, &toolbox.PP[i].to);
    scanf("%d", &nPipe);
    for (i = 0; i < nPipe; i++)
      scanf("%d%d", &toolbox.PP[nPtr+i].from, &toolbox.PP[nPtr+i].to);
    toolbox.size = nPtr + nPipe;

    printf("%d\n", getStep(nGrid));
  }

  return 0;
}
```

5. 输出结果

运行基于图 10.12 建立的管道,求解结果如下。

```
3
4
```

本章小结

本章主要介绍了基于数据结构原理的部分经典问题求解,主要学习要点如下。

- 理解递归算法的分类及设计思想,掌握经典递归问题的算法实现。
- 理解图遍历算法的基本思想,掌握经典遍历问题的算法实现。

附录A 全国计算机专业数据结构考研大纲

一、数据结构和算法的基本概念

1. 数据结构的基本概念，包括逻辑结构、存储结构的基本概念、两者之间的区别与联系。
2. 算法的基本概念和性质。
3. 算法时间复杂度的基本概念，注意递归算法的复杂度计算方法。

二、线性表

1. 线性表的逻辑结构定义和基本操作。
2. 顺序表结构实现和基本操作实现。
3. 链表结构实现和基本操作实现。
4. 顺序表和链表一般应用问题操作实现。

三、栈和队列

1. 栈的基本概念和性质。
2. 顺序栈结构实现和基本操作实现。
3. 链栈结构实现和基本操作实现。
4. 栈与递归的关系，实现递归算法的转换。
5. 队列的基本概念和性质。
6. 顺序队列结构实现和基本操作实现。
7. 链队列结构实现和基本操作实现。
8. 栈和队列的应用。

四、串

1. 串的基本概念。
2. 顺序串和链串结构实现。
3. 串的模式匹配操作实现(BF 算法和 KMP 算法)。

五、数组和广义表

1. 数组(二维数组和三维数组)的顺序存储实现。
2. 特殊矩阵(对称矩阵、三角矩阵、对角矩阵、稀疏矩阵)的压缩存储实现。

六、树和二叉树

1. 树的基本概念和性质。
2. 二叉树的基本概念和性质。
3. 二叉树的存储结构实现(顺序存储结构、链式存储结构)。
4. 二叉树的遍历操作实现(先序遍历、中序遍历、后序遍历、层次遍历)。
5. 线索二叉树的基本概念和构造。
6. 树、森林和二叉树的关系及转换。
7. 树的存储结构实现。
8. 哈夫曼树的基本概念和构造。
9. 哈夫曼编码的基本概念和计算。
10. 树与二叉树的基本应用。

七、图

1. 图的基本概念。
2. 图的存储结构实现(邻接矩阵、邻接表)
3. 图的遍历操作实现(DFS、BFS)。
4. 最小生成树的基本概念和构造(选点法、选边法)。
5. 拓扑排序过程实现。
6. 关键路径的基本概念和求解计算。
7. 最短路径算法实现(单源最短路径、全源最短路径)。

八、查找

1. 查找的基本概念。
2. 顺序查找过程实现。
3. 折半查找过程实现。
4. 二叉排序树的基本概念和性质。
5. 二叉排序树上的查找、插入、删除算法实现。
6. 平衡二叉树的基本概念及调整算法。
7. B—树的基本概念及基本操作(插入关键字、删除关键字)。
8. B+树的基本概念。
9. 哈希表、哈希方法的基本概念。
10. 哈希查找过程实现。
11. 各种查找算法的分析与应用。

九、内排序

1. 排序的基本概念。
2. 插入排序(直接插入排序、希尔排序)算法思想和实现。

3. 选择排序(简单选择排序、堆排序)算法思想和实现。
4. 交换排序(冒泡排序、快速排序)算法思想和实现。
5. 二路归并排序算法思想和实现。
6. 基数排序算法思想和实现。
7. 各种排序算法的分析与应用。

参 考 文 献

[1] 严蔚敏,吴伟民.数据结构(C语言版)[M].北京:清华大学出版社,1997.
[2] 李春葆,等.数据结构教程[M].4版.北京:清华大学出版社,2010.
[3] 刘大有,等.数据结构联考辅导教程(2011版)[M].北京:清华大学出版社,2010.
[4] 黄杨铭,等.数据结构[M].北京:科学出版社,2001.
[5] 黄刘生.数据结构[M].北京:经济科学出版,2000.
[6] 许卓群,等.数据结构与算法[M].北京:高等教育出版社,2004.
[7] 殷人昆,等.数据结构教程(面向对象方法与C++描述)[M].北京:清华大学出版社,1999.
[8] 王庆瑞.数据结构——C语言版[M].北京:北京希望电子出版社,2002.
[9] 陈守礼,等.算法与数据结构C语言版[M].北京:机械工业出版社,2008.
[10] 王宏生,宋继红.数据结构[M].北京:国防工业出版社,2006.
[11] 左飞.C++数据结构原理与经典问题求解[M].北京:电子工业出版社,2008.
[12] P J Plauger,A S Alexander,Meng Lee. C++ STL中文版[M].王昕,译.北京:中国电力出版社,2002.
[13] 刘汝佳,黄亮.算法艺术与信息学竞赛[M].北京:清华大学出版社,2003.
[14] 吴昊.ACM程序设计培训教程[M].北京:中国铁道出版社,2007.
[15] 傅彦,等.离散数学及其应用[M].2版.北京:高等教育出版社,2013.
[16] 徐孝凯.数据结构[M].北京:电子工业出版社,2004.
[17] 张选平,雷永梅.数据结构[M].北京:机械工业出版社,2002.

图书资源支持

感谢您一直以来对清华版图书的支持和爱护。为了配合本书的使用，本书提供配套的资源，有需求的读者请扫描下方的“书圈”微信公众号二维码，在图书专区下载，也可以拨打电话或发送电子邮件咨询。

如果您在使用本书的过程中遇到了什么问题，或者有相关图书出版计划，也请您发邮件告诉我们，以便我们更好地为您服务。

我们的联系方式：

地　　址：北京市海淀区双清路学研大厦 A 座 701

邮　　编：100084

电　　话：010-62770175-4608

资源下载：http://www.tup.com.cn

客服邮箱：tupjsj@vip.163.com

QQ：2301891038（请写明您的单位和姓名）

资源下载、样书申请



书圈



扫一扫，获取最新目录

用微信扫一扫右边的二维码，即可关注清华大学出版社公众号“书圈”。